高等学校计算机规划教材

C++程序设计基础

幸莉仙　主编

于海泳　王立军　田志刚　王华英　编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书针对初学者学习程序设计而编写，通过本书的学习，初学者可以较好地掌握结构化程序设计的3种结构、面向对象的概念和编程思想。本书以 VC++ 2005 为开发平台，结合大量实例，系统本书介绍 VC++ 2005 的开发环境、基本语法和编程技巧。全书共 11 章：C++与 VC++ 2005 概述，VC++ 2005 程序设计基础，流程控制语句，数组和字符串，指针，函数，结构体与联合，类与对象，类的继承、派生与多态，C++流与文件操作，VC++ 2005 应用程序开发实例。本书配有电子课件、源代码等教学资源。

本书可作为普通高等学校 C++程序设计的教学用书，也可作为计算机等级考试的培训教材和 VC++ 2005 的自学用书。

图书在版编目(CIP)数据

C++程序设计基础 / 幸莉仙主编. —北京：电子工业出版社，2011.1
高等学校计算机规划教材
ISBN 978-7-121-12226-2

Ⅰ. ①C… Ⅱ. ①幸… Ⅲ. ①C 语言－程序设计－高等学校－教材 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2010)第 217099 号

策划编辑：史鹏举
责任编辑：史鹏举
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：16　　字数：410 千字
印　　次：2011 年 1 月第 1 次印刷
定　　价：28.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010)88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010)88258888。

前　言

网络时代学习网络版的软件开发工具非常必要。“短视近利，够用就好”不是学习者应有的态度和方法，未来对软件开发环境有太多的需求和挑战，本书采用 Visual Studio 2005.NET 作为学习 C++的语言环境正是为了贴近人才市场对学生能力和技能需要。

C++语言是当前最流行、最实用的一种计算机高级语言，尽管国内大多数高校都把 C++设定为新生计算机编程知识入门类课程，但长期以来却在讲授单机版的 Turbo C 或 VC++ 6.0，学生学完后，不会使用目前流行的.NET 开发工具编写应用程序。本书将面向 Visual Studio 2005.NET 的初中级用户，系统地介绍 C++程序开发的基础知识、编程方法和技巧。

本书经过精心规划，具有以下特点：

1. 从 VC++ 2005 最基本的数据类型、概念、语法、简单程序编写开始入手，使读者逐步掌握结构化程序设计的三种基本结构，即顺序结构、选择结构、循环结构；通过介绍面向对象的概念以及在 VC++ 2005 环境下的实现，使读者从基本概念、基础操作的学习上升到对理论的理解，从而领会应用程序开发的实质。

2. 在例题选择上秉承由浅入深、由简到繁的编程规律，对典型例题给出多种算法求解，在习题上力求做到多样化，以培养和提高初学者分析问题和解决问题的能力。

3. VC++ 2005 语言系统庞大，知识点前后衔接紧密，为使初学者轻松学习本门课程，掌握程序设计的精髓，本书将所有知识点按章节划分成一个有序的线性结构，内容由易到难、循序渐进。

4. 为使初学者在学习完本课程后能编写出完成的 Windows 应用程序，在第11章介绍了一个完整的应用程序开发实例。

5. 所有例题、习题、函数等程序都在 VC++ 2005 环境下测试通过。

6. 附录给出了编程中常用的库函数(包括数学函数、字符串函数、常用数学函数的反函数等)，以及习题答案，以方便读者学习时使用。

本书共分 11 章，依次是：C++与 VC++ 2005 概述，VC++ 2005 程序设计基础，流程控制语句，数组和字符串，指针，函数，结构体与联合，类与对象，类的继承、派生与多态，C++流与文件操作，VC++ 2005 应用程序开发实例。各章之间内容衔接紧密、自然，形成了一个完整的学习体系。

本书配有电子课件、源代码等教学资源，需要者可登录华信教育资源网 http://www.hxedu.com.cn，免费注册下载。

本书由幸莉仙策划和统稿，由幸莉仙、于海泳、王立军、田志刚、王华英共同撰写完成。韩玢、黄慧莲对本书进行了排版和校对工作。

由于时间仓促，书中难免有一些疏漏和不足，恳请广大读者和同行不吝赐教，以便及时修订和补充。

联系方式：0312-7525111。E-mail：xlxwhy@sina.com.cn

编　者

目　录

第 1 章　C++与 VC++ 2005 概述 ····· (1)
1.1　计算机程序设计语言的发展 ····· (1)
1.1.1　机器语言 ····· (1)
1.1.2　汇编语言 ····· (1)
1.1.3　高级语言 ····· (2)
1.1.4　结构化程序设计语言 ····· (2)
1.1.5　面向对象语言的产生 ····· (3)
1.2　C++语言与面向对象程序设计 ····· (4)
1.2.1　C++概述 ····· (4)
1.2.2　面向对象程序设计 ····· (4)
1.3　C++集成开发环境 Visual Studio 2005 ····· (7)
1.3.1　集成开发环境 IDE ····· (7)
1.3.2　Visual Studio 2005 简介 ····· (7)
1.4　简单的 VC++ 2005 程序 ····· (8)
1.4.1　VC++ 2005 程序的开发过程 ····· (8)
1.4.2　简单的 VC++ 2005 程序示例 ····· (9)
本章小结 ····· (13)
习题 1 ····· (13)

第 2 章　VC++ 2005 程序设计基础 ····· (15)
2.1　VC++ 2005 基本语法 ····· (15)
2.1.1　字符集 ····· (15)
2.1.2　词法记号 ····· (15)
2.2　基本数据类型和表达式 ····· (18)
2.2.1　基本数据类型 ····· (18)
2.2.2　字面常量 ····· (19)
2.2.3　变量 ····· (22)
2.2.4　符号常量 ····· (24)
2.2.5　运算符与表达式 ····· (24)
2.2.6　语句 ····· (32)
2.3　数据的输入与输出 ····· (32)
2.3.1　I/O 流 ····· (32)
2.3.2　预定义的插入符和提取符 ····· (33)
2.3.3　简单的 I/O 格式控制 ····· (33)
2.4　基于 VC++ 2005 的简单程序开发 ····· (34)

2.4.1 一个简单程序设计例程 ····· (34)
2.4.2 main 函数 ····· (35)
2.4.3 注释 ····· (36)
2.4.4 编译预处理 ····· (36)
2.4.5 命名空间与 using 应用 ····· (40)
本章小结 ····· (42)
习题 2 ····· (43)

第 3 章 流程控制语句 ····· (46)
3.1 程序的基本控制结构 ····· (46)
3.1.1 语句的分类 ····· (46)
3.1.2 结构化程序控制结构 ····· (47)
3.2 流程控制语句 ····· (47)
3.2.1 if 语句 ····· (47)
3.2.2 switch 语句 ····· (52)
3.3 循环控制语句 ····· (54)
3.3.1 for 循环 ····· (54)
3.3.2 do while 循环 ····· (56)
3.3.3 while 循环 ····· (58)
3.4 跳转语句 ····· (59)
3.4.1 break 语句 ····· (59)
3.4.2 continue 语句 ····· (60)
3.4.3 goto 语句 ····· (61)
3.4.4 return 语句 ····· (62)
本章小结 ····· (62)
习题 3 ····· (62)

第 4 章 数组和字符串 ····· (65)
4.1 数组的概念 ····· (65)
4.2 数组的定义和数组元素表示方法 ····· (65)
4.2.1 数组的定义 ····· (66)
4.2.2 格式举例 ····· (67)
4.3 数组元素的输入与输出 ····· (67)
4.4 数组的应用 ····· (69)
4.4.1 统计 ····· (70)
4.4.2 排序 ····· (71)
4.4.3 查找 ····· (72)
4.4.4 数组的其他应用 ····· (74)
4.5 字符串 ····· (76)
4.5.1 字符串的概念 ····· (76)
4.5.2 字符串函数 ····· (78)

4.5.3 字符串应用举例 ····· (80)
本章小结 ····· (82)
习题 4 ····· (82)

第 5 章 指针 ····· (85)
5.1 指针的概念 ····· (85)
5.2 指针变量 ····· (85)
5.3 指针运算 ····· (86)
5.4 指针与数组 ····· (88)
5.4.1 指针与一维数组 ····· (88)
5.4.2 指针与二维数组 ····· (90)
5.4.3 new 与 delete ····· (91)
5.5 引用变量 ····· (92)
本章小结 ····· (94)
习题 5 ····· (94)

第 6 章 函数 ····· (97)
6.1 函数的定义与调用 ····· (97)
6.1.1 函数的定义 ····· (97)
6.1.2 函数的声明与调用 ····· (99)
6.2 函数调用方式和参数传递 ····· (101)
6.2.1 函数调用过程 ····· (101)
6.2.2 传值调用 ····· (101)
6.2.3 传址调用 ····· (102)
6.2.4 数组作为参数调用 ····· (103)
6.3 变量的作用域 ····· (105)
6.3.1 作用域分类 ····· (106)
6.3.2 应用举例 ····· (107)
6.4 递归函数 ····· (109)
6.5 重载函数 ····· (112)
6.6 模板函数 ····· (113)
6.7 内联函数 ····· (116)
6.8 函数指针 ····· (117)
本章小结 ····· (121)
习题 6 ····· (121)

第 7 章 结构体与联合 ····· (124)
7.1 结构体类型 ····· (124)
7.1.1 结构体的定义 ····· (124)
7.1.2 结构体变量的定义和初始化 ····· (125)
7.1.3 结构体变量的引用 ····· (126)
7.1.4 结构体数组 ····· (128)

7.1.5 结构体与函数 ····· (130)
7.1.6 结构体指针 ····· (133)
7.1.7 结构体与链表 ····· (137)
7.2 联合 ····· (139)
7.2.1 联合的定义 ····· (139)
7.2.2 联合变量的定义 ····· (140)
7.2.3 联合变量的引用 ····· (142)
7.3 枚举类型 ····· (143)
7.4 结构体与联合应用实例 ····· (146)
本章小结 ····· (148)
习题 7 ····· (148)

第 8 章 类与对象 ····· (150)
8.1 类的概念与定义 ····· (150)
8.1.1 面向对象程序设计概述 ····· (150)
8.1.2 类的声明 ····· (155)
8.1.3 类的成员函数 ····· (157)
8.1.4 类与结构体 ····· (158)
8.2 对象 ····· (159)
8.2.1 对象的定义 ····· (159)
8.2.2 对象成员的引用 ····· (160)
8.3 构造函数 ····· (161)
8.3.1 构造函数的作用 ····· (161)
8.3.2 带参数的构造函数 ····· (163)
8.3.3 构造函数重载 ····· (164)
8.3.4 拷贝构造函数 ····· (166)
8.4 析构函数 ····· (167)
8.5 类的静态成员 ····· (168)
8.5.1 静态数据成员 ····· (169)
8.5.2 静态成员函数 ····· (170)
8.6 友元 ····· (172)
8.6.1 友元函数 ····· (172)
8.6.2 友元类 ····· (174)
8.7 VC++ 2005 中使用类向导 ····· (175)
本章小结 ····· (178)
习题 8 ····· (178)

第 9 章 类的继承、派生与多态 ····· (181)
9.1 类的继承与派生 ····· (181)
9.1.1 继承与派生的概念 ····· (181)
9.1.2 派生类定义的格式 ····· (182)

9.1.3 继承方式 (186)
9.1.4 多重继承 (192)
9.2 多态与虚函数 (194)
9.2.1 多态的概念 (194)
9.2.2 虚函数 (197)
9.2.3 多态的实现机制 (197)
9.2.4 纯虚函数与抽象类 (199)
本章小结 (201)
习题 9 (202)
第 10 章 C++流与文件操作 (205)
10.1 C++流的概念 (205)
10.2 输入/输出标准流类 (205)
10.2.1 C++中的 I/O 流库 (205)
10.2.2 标准输入/输出流对象 (205)
10.3 文件操作 (210)
10.3.1 文件的打开与关闭 (210)
10.3.2 文本文件的读写操作 (211)
10.3.3 二进制文件的读写操作 (213)
10.4 应用举例 (216)
本章小结 (220)
习题 10 (220)
第 11 章 VC++ 2005 应用程序开发实例 (223)
11.1 MFC 应用程序 (223)
11.1.1 创建应用程序 (223)
11.1.2 应用程序的运行 (224)
11.1.3 应用程序类和源文件 (225)
11.1.4 应用程序的控制流程 (226)
11.2 调用 Windows 公共对话框的实例 (227)
11.2.1 使用对话框编辑器 (227)
11.2.2 编写代码 (228)
11.3 利用 VC++ 2005 连接数据库实例 (230)
11.3.1 建立工程 DAOAccess (230)
11.3.2 建立 Access 文件 (230)
11.3.3 修改主窗体界面 (230)
11.3.4 添加代码 (231)
附录 A ASCII 码表 (234)
附录 B 习题答案 (236)
附录 C 常用库函数 (239)
附录 D 程序调试与异常处理 (242)

第 1 章　C++与 VC++ 2005 概述

1.1　计算机程序设计语言的发展

自 1946 年第一台电子计算机问世以来，计算机已被广泛地应用于生产、生活的各个领域，推动着社会的进步与发展。特别是 Internet 出现后，传统的信息收集、传输及交换方式发生了革命性的改变。

计算机科学的发展依赖于计算机硬件和软件技术的发展，硬件是计算机的躯体，软件是计算机的灵魂。没有软件，计算机只是一台“裸机”，什么也不能干；有了软件，计算机才有“思想”，才能做相应的事。软件是用计算机程序设计语言编写的。计算机程序设计语言的发展经历了从机器语言、汇编语言到高级语言的历程，如图1-1 所示。

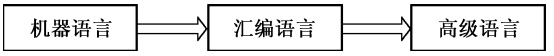


图 1-1　计算机语言发展历程

1.1.1　机器语言

计算机使用的是由“0”和“1”组成的二进制数，二进制编码方式是计算机语言的基础。计算机发明之初，科学家只能用二进制数编制的指令控制计算机运行。每一条计算机指令均由一组“0”、“1”数字按一定的规则排列组成，若要计算机执行一项简单的任务，需要编写大量的这种指令。这种有规则的二进制数组成的指令集，就是机器语言(Machine Language)(也称指令系统)。不同系列的 CPU，具有不同的机器语言，如目前个人计算机中常用 AMD 公司的 CPU 系列和 Intel 公司的 CPU 系列，具有不同的机器语言。

机器语言是计算机唯一能识别并直接执行的语言，与汇编语言或高级语言相比，其执行效率高。但其可读性差，不易记忆；编写程序既难又繁，容易出错；程序调试和修改难度巨大，不容易掌握和使用。此外，因为机器语言直接依赖于中央处理器，所以用某种机器语言编写的程序只能在相应的计算机上执行，无法在其他型号的计算机上执行，也就是说，可移植性差。

1.1.2　汇编语言

为了减轻使用机器语言编程的痛苦，20 世纪 50 年代初，出现了汇编语言(Assemble Language)。汇编语言用比较容易识别、记忆的助记符替代特定的二进制串。下面是几条 Intel 80x86 的汇编指令：

```
ADD AX, BX;      表示将寄存器 AX 和 BX 中的内容相加，结果保存在寄存器 AX 中。
SUB AX, NUM;     表示将寄存器 AX 中的内容减去 NUM，结果保存在寄存器 AX 中。
MOV AX, NUM;     表示把数 NUM 保存在寄存器 AX 中。
```

通过这种助记符，人们就能较容易地读懂程序，调试和维护也更方便了。但这些助记符号计算机无法识别，需要一个专门的程序将其翻译成机器语言，这种翻译程序称为汇编程序。

汇编语言的一条汇编指令对应一条机器指令，与机器语言性质上是一样的，只是表示方式做了改进，其可移植性与机器语言一样不好。总之，汇编语言是符号化的机器语言，执行效率仍接近于机器语言，因此，汇编语言至今仍是一种常用的软件开发工具。

1.1.3 高级语言

尽管汇编语言比机器语言方便，但汇编语言仍然具有许多不便之处，程序编写的效率远远不能满足需要。1954 年，第一个高级语言 FORTRAN 问世了。高级语言(High-level Language)是一种用能表达各种意义的“词”和“数学公式”按一定的“语法规则”编写程序的语言，也称为高级程序设计语言或算法语言。半个多世纪以来，有几百种高级语言问世，影响较大、使用较普遍的有 FORTRAN、A LGOL、COBOL、BASIC、LISP、SNOBOL、PL/1、Pascal、C、PROLOG、Ada、C++、Visual C++、Visual Basic 、Delphi、Java、C#等。高级语言的发展也经历了从早期语言到结构化程序设计语言、面向对象程序设计语言的过程。

高级语言与自然语言和数学表达式相当接近，不依赖于计算机型号，通用性较好。高级语言的使用，大大提高了程序编写的效率和程序的可读性。

与汇编语言一样，计算机无法直接识别和执行高级语言，必须翻译成等价的机器语言程序(称为目标程序)才能执行。高级语言源程序翻译成机器语言程序的方法有“解释”和“编译”两种。解释方法采用边解释边执行的方法，如早期的 BASIC 语言即采用解释方法，在执行 BASIC 源程序时，解释一条 BASIC 语句，执行一条语句。编译方法采用相应语言的编译程序，先把源程序编译成指定机型的机器语言目标程序，然后再把目标程序和各种标准库函数连接装配成完整的目标程序，在相应的机型上执行。如 C、C++、Visual C++及 Visual Basic 等均采用编译的方法。编译方法比解释方法更有效率。

1.1.4 结构化程序设计语言

高级语言编写程序的编写效率虽然比汇编语言高，但随着计算机硬件技术的日益发展，人们对大型、复杂的软件需求量剧增，同时因缺乏科学规范、系统规划与测试，程序含有过多错误而无法使用，甚至带来巨大损失。20 世纪 60 年代中后期“软件危机”的爆发，使人们认识到大型程序的编制不同于小程序。“软件危机”的解决一方面需要对程序设计方法、程序的正确性和软件的可靠性等问题进行深入研究，另一方面需要对软件的编制、测试、维护和管理方法进行深入研究。

1968 年，E. W. Dijkstra 首先提出“GOTO 语句有害”论点，引起了人们对程序设计方法讨论的普遍重视。程序设计方法学在这场讨论中逐渐产生和形成。程序设计方法学是一门讨论程序性质、设计理论和方法的学科。它包含的内容比较丰富，如结构化程序设计、程序的正确性证明、程序变换、程序的形式说明与推导，以及自动程序设计等。

在程序设计方法学中，结构化程序设计(Structural Programming)占有重要的地位，可以说，程序设计方法学是在结构化程序设计的基础上逐步发展和完善的。结构化程序设计是一种程序设计的原则和方法，它讨论了如何避免使用 GOTO 语句；如何将大规模、复杂的流程图转换成一种标准的形式，使得它们能够用几种标准的控制结构(顺序、分支和循环)通过重复和嵌套来表示。结构化程序设计思想采用“自顶向下、逐步求精”的方法，避免被具体的细节所缠绕，降低难度，直到恰当的时机才考虑实现的细节，从而有效地将复杂的程序系统设计任务分解成许多易于控制和处理的子程序，便于开发和维护。

按结构化程序设计的要求设计出的高级程序设计语言称为结构化程序设计语言。利用结构化程序设计语言，或者说按结构化程序设计思想编写出来的程序，称为结构化程序。结构化程序具有结构清晰、容易理解、容易修改、容易验证等特点。

到了 20 世纪 70 年代末期，随着计算机应用领域的不断扩大，对软件技术的要求越来越高，结构化程序设计语言和结构化程序设计方法也无法满足用户需求的变化，其缺点日益显露出来：

(1) 代码的可重用性差。随着软件规模的逐渐庞大，代码重用成了提高程序设计效率的关键；但采用传统的结构化设计模式，程序员每进行一个新系统的开发，几乎都要从零开始，这中间需要做大量重复、繁琐的工作。

(2) 可维护性差。结构化程序是由大量的过程(函数、子程序)组成的；随着软件规模逐渐庞大，程序变得越来越复杂，过程(函数、子程序)越来越多，相互间的耦合越来越高，它们变得难以管理；当某个业务有所变化时必须对大量的程序进行修改和调试。

(3) 稳定性差。结构化程序要求模块独立，并通过过程(函数、子程序)的概念来实现。但这一概念狭隘、稳定性有限，在大型软件开发过程中，数据的不一致性问题仍然存在。

(4) 难以实现。在结构化程序中，代码和数据是分离的。例如，在 C 语言中，代码单位为函数，而数据单位称为结构，函数和结构没有结合在一起。然而，函数和数据结构并不能充分地模拟现实世界。例如，当考虑会计部门的应用程序时，应该考虑下列内容：

① 出纳支付工资。

② 职工出具凭证。

③ 财务主管批准支付。

④ 出纳记账。

但实际应用中，要决定如何通过数据结构、变量和函数来实现这个应用程序却是很困难的。

1.1.5　面向对象语言的产生

结构化程序设计方法与语言是面向过程的，存在较多的缺点，同时程序的执行是流水线式的，在一个模块被执行完成前不能干别的事，也无法动态地改变程序的执行方向。这和人们日常认识、处理事物的方式不一致。人们认为客观世界是由各种各样的对象(或称实体、事物)组成的；每个对象都有自己的内部状态和运动规律，不同对象间的相互联系和相互作用构成了各种不同的系统，进而构成整个客观世界；计算机软件主要是为了模拟现实世界中的不同系统，如物流系统、银行系统、图书管理系统、教学管理系统等。因此，计算机软件可以认为是，现实世界中相互联系的对象所组成的系统在计算机中的模拟实现。

为了使计算机更易于模拟现实世界，1967 年挪威计算中心的 Kisten.Nygaard 和 Ole.Johan Dahl 开发了 Simula67 语言，它提供了比子程序更高一级的抽象和封装，引入了数据抽象和类的概念，被认为是第一个面向对象(Object Oriented)程序设计语言。20 世纪 70 年代初，Palo Alto 研究中心的 Alan Kay 所在的研究小组开发出了 Smalltalk 语言，之后又开发出了 Smalltalk-80。Smalltalk-80 被认为是最纯正的面向对象语言，它对后来出现的面向对象语言，如 Object-C、C++、Java、Self、Eiffl 产生了深远的影响。

随着面向对象语言的出现，面向对象程序设计方法也应运而生且得到迅速发展，面向对象的思想也不断向其他方面渗透。1980 年 Grady Booch 提出了面向对象设计的概念，之后面向对象分析的概念也被提出。1990 年以来，面向对象分析、测试、度量和管理等研究得到了

长足的发展，并在全世界掀起了一股面向对象热潮，至今盛行不衰。面向对象程序设计在软件开发领域掀起了巨大的变革，极大地提高了软件开发效率。

1.2 C++语言与面向对象程序设计

1.2.1 C++概述

当 C 语言程序代码达到 25 000 行以上后，维护和修改工作变得相当困难。为了满足管理程序复杂性的需要，美国贝尔实验室的 Bjarne Stroustrup 博士于 1979 年开始对 C 语言进行了改进和扩充，并引入了面向对象程序设计的内容，1983 年命名为 C++，后经过三次重大修订，于 1994 年制定了标准 C++草案，之后经过不断完善，成为目前的 C++。

C++提出了一些更为深入的概念，它所支持的这些面向对象的概念容易将问题空间直接地映射到程序空间，为程序员提供了一种与传统结构程序设计不同的思维方式和编程方法。因而也增加了整个语言的复杂性，掌握起来有一定难度。

C++具有以下特点：

⑴ 保持了与 C 语言的兼容性。绝大多数 C 语言程序不经修改可以直接在 C++环境中运行。

⑵ 支持面向过程的程序设计。它是一种理想的结构化程序设计语言，又包含了面向对象程序设计的特征。C++由两部分组成：一是过程性语言部分，与 C 语言无本质区别；二是类和对象部分，是面向对象程序设计的主体。

⑶ 具有程序效率高、灵活性强的特点。C++使程序结构清晰、易于扩展、易于维护而不失效率。

⑷ 具有通用性和可移植性。C++是一种标准化的、与硬件基本无关的程序设计语言，C++程序通常无须修改或稍许修改便可在其他计算机上运行。

⑸ 具有丰富的数据类型和运算符，并提供了强大的库函数。

⑹ 具有面向对象的特性，C++支持抽象性、封装性、继承性和多态性。

1.2.2 面向对象程序设计

面向过程程序设计缺点的根源在于数据与数据处理分离，而面向对象程序设计（Object Oriented Programming）方法正是克服这个缺点，同时吸纳结构化设计思想的合理部分而发展起来的，这两种设计思想并非对立关系。面向对象设计思想模拟自然界认识和处理事物的方法，将数据和对数据的操作方法放在一起，形成一个相对独立的整体——对象（Object），对同类型对象抽象出共性，形成类（Class）。任何一个类中的数据都只能用本类自有的方法进行处理，并通过简单的接口与外部联系。对象之间通过消息（Message）进行通信。以下就面向对象程序设计中的基本概念、面向对象软件开发方法和面向对象程序设计的特点做简单介绍。

⑴ 对象

在自然界中，对象是一个常见概念，一般将所面对的事物看做具有某些属性和行为（或操作）的对象。例如，手表是一个对象，它具有的属性包括表针、旋钮及复杂的机械结构等，行为包括调节旋钮这样的操作。其中调节旋钮提供了对外的接口。在手表之外，只能通过这个接口对对象进行操作，而不能直接调整其内部的机械零件。在面向对象程序设计中，同样

使用对象的概念描述问题和事物，但更加抽象，含义的范围也更加广泛，可包括有形实体、图形，甚至抽象概念，如程序设计中的窗口、单击鼠标这样的事件等，都可以抽象成为一个对象。对象是组成程序系统的基本单位。

(2) 类

人类在认识事物时，通常是分类进行的。通过抽象的方法，从一些对象中概括出此类对象共有的静态特征(属性)和动态特征(行为)，形成类。类是一个抽象的概念，用来描述这类对象所共有的、本质的属性和行为。任何一个对象都是这个类的一个具体实现，称为实例(instance)。同类对象之间具有相同的属性和行为。类和对象之间的关系正如手表和一块具体的手表之间的关系。手表只是个概念，这个概念描述了所有手表共有的属性(包括表针、旋钮、内部结构)和行为(调节旋钮)，而一块具体的手表则是一个实在的对象。两块手表可能外形不同，但都具有手表所共有的属性和操作。

面向对象程序设计也使用分类抽象的方法，通过对一类对象的抽象形成“类”。在程序设计中，“类”表现为一种用户定义的数据类型，用这种数据类型描述该类所具有的属性和行为，其中属性用数据进行描述，而行为用一系列函数描述，也称操作。例如，定义一个矩形类，它的数据包括矩形的顶点坐标，而方法可包含以下函数：移动矩形位置、扩大、缩小等。如用这个矩形类定义两个矩形对象，这两个矩形对象就是同类对象，它们都用顶点坐标来描述，都可以进行移动、扩大和缩小等操作。

(3) 消息

自然界是由各种各样的对象组成的，这些对象之间通过信息传递产生相互作用，构成富有生机的世界。人类将对象之间产生相互作用所传递的信息称做消息。例如，汽车和人是两个对象，人起动汽车，就是向汽车发送消息，转动方向盘，也是发送消息，其中转动的角度是消息中的参数。汽车接收到消息后，按照消息及其参数执行相应的操作。

在面向对象设计的程序中，对象之间的相互作用也是通过消息机制实现的。

(4) 面向对象的软件开发方法

在面向对象程序设计发展的早期，软件业界主要集中于研究面向对象的编程(OOP)，但对于大型软件开发过程，编程只是其中一个很小的部分。面向对象方法的根本合理性在于它符合客观世界的组成方式和大脑的思维方式，因此面向对象的思想方法应贯穿软件开发的全过程，这就是面向对象的软件工程。

面向对象的软件工程同样遵循分层抽象、逐步细化的原则，软件开发过程包括面向对象的分析(OOA, Object Oriented Analysis)、面向对象的设计(OOD, Object Oriented Design)、面向对象的编程(OOP, Object Oriental Programming)、面向对象的测试(OOT, Object Oriented Test)、面向对象的维护(OOSM, Object Oriented Soft Maintenance)五个阶段。

分析阶段的主要任务是按照面向对象的概念和方法，从问题中识别出有意义的对象，以及对象的属性、行为和对象间的通信，进而抽象出类结构，最终将它们描述出来，形成一个需求模型，由于系统的复杂性，这个模型一般只能反映用户对系统主体部分的需求。

设计阶段从需求模型出发，分别进行类的设计和应用程序的设计。类的设计需要应用分层抽象的方法，而应用程序设计是根据已设计好的类来构造满足要求的应用程序。

编程阶段实现由设计表示到面向对象程序设计语言描述的转换。

测试的任务在于发现并改正程序中的错误。面向对象程序设计中，类是程序的基本单元，因此也是测试的基本单元。经过测试后的程序进入运行维护期，即投入使用。

⑸ 面向对象程序设计的特点

面向对象程序设计中，对象是程序的基本单元。从类和对象的概念及面向对象设计方法所提供的支持看，这种设计方法具有以下几个特点及相应的优点。

封装性(Ecapsulation)：对象是一个封装体，在其中封装了该对象所具有的属性和操作。对象作为独立的基本单元，实现了将数据和数据处理相结合的思想。此外，封装特性还体现在可以限制对象中数据和操作的访问权限，从而将属性“隐藏”在对象内部，对外只呈现一定的外部特性和功能。手表是一个典型的例子，大量的零件和动作被封装在外壳中，并被隐藏起来，提供给我们的只能是读表盘和旋旋钮。同样，可以将一个对象中的数据隐藏起来，而方法定义为开放，那么在这个类之外，不能直接引用其中的数据，只能通过方法达到间接使用数据的目的。就好比不能直接去拨驱动表针的内部结构，只能通过旋旋钮实现一样。

封装性增加了对象的独立性，C++通过建立数据类型——类，来支持封装和数据隐藏。一个定义完好的类一旦建立，就可看成完全的封装体，作为一个整体单元使用，用户不需要知道这个类是如何工作的，而只需要知道如何使用就行。另一方面，封装增加了数据的可靠性，保护类中的数据不被类以外的程序随意使用。这两个优点十分有利于程序的调试和维护。

继承(Inheritance)和派生性：以汽车为例，如果已经定义了汽车类，现在需要定义小汽车，通常我们不会重复描述属于汽车的那些共有特征，而是在继承汽车类特性的基础上，描述出属于小汽车的新的特征。于是称小汽车继承了汽车，也可以称是由汽车派生出来的。面向对象程序设计提供了类似的机制。当定义了一个类后，又需定义一个新类，这个新类与原来类相比，只是增加了或修改了部分属性和操作，这样在定义新类时只需说明新类继承原来类，然后描述出新类所特有的属性和操作即可。此时称原来类为基类，由它派生出来的类称为子类或派生类。由基类派生出子类，子类还可继续派生它的子类，如此下去，可以形成树状派生关系，称为派生树或继承树；如图1-2所示(注意箭头指向基类)。

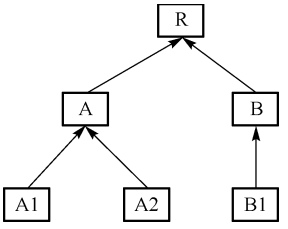


图 1-2 类的继承关系

继承性可以简化人们对问题的认识和描述，同时还可以在开发新程序和修改源程序时最大限度利用已有的程序，提高了程序的可重用性，从而提高了程序修改、扩充和设计的效率。

多态性(Polymorphism)：多态性是指同样一个消息，被不同对象接收时，产生不同的结果。系统提供的这种机制主要用在具有继承关系的类体系中。一个类体系中的不同对象可以用不同方式响应同一消息，并产生不同结果，即实现“同一接口，多种方法”。例如，定义了中学生类，再由中学生派生出大学生类。对于“计算平均成绩”这样一个消息，对中学生对象，计算的是语文、数学、英语等课程；而对大学生对象，则计算高等数学、英语、线性代数等课程。

继承和多态性的组合，可以很容易地生成一系列虽然类似但独一无二的对象。继承性使这些对象共享许多相似特征，而多态性使同样的操作对不同的对象有不同的表现方式。这样既提高了程序的灵活性，又减轻了分别逐个设计的负担。

面向对象程序设计着眼点是对象，程序设计的核心是从问题中抽象出合适的对象，即首先解决“做什么”的问题。至于“怎么做”，则封装在对象内部。而对于操作方法的设计，核心仍然是算法的设计，完全吸收了结构化程序设计的思想。

1.3　C++集成开发环境 Visual Studio 2005

1.3.1　集成开发环境 IDE

集成开发环境(Integrated Development Environment，IDE)是用于提供程序开发环境的应用程序，一般包括代码编辑器、编译器、调试器和图形用户界面工具。就是集成了代码编写功能、分析功能、编译功能、调试功能等一体化的开发软件服务套。所有具备这一特性的软件或软件套(组)都可以叫做集成开发环境。如微软的 Visual Studio 系列，Borland 的 C++ Builder、Delphi 系列等。该程序可以独立运行，也可以和其他程序并用。例如，BASIC 语言在微软办公软件中可以使用，可以在微软 Word 文档中编写 Word Basic 程序。IDE 为用户使用 Visual Basic、Java 和 PowerBuilder 等现代编程语言提供了方便。

不同的技术体系有不同的 IDE。比如 Visual Studio.Net 可以称为 C++、VB、C#等语言的集成开发环境，所以 Visual Studio.Net 可以叫做 IDE。同样，Borland 的 JBuilder 也是一个 IDE，它是 Java 的 IDE。Zend Studio、Editplus、Ultraedit，它们都具备基本的编码、调试功能，所以都可以称做 IDE。

1.3.2　Visual Studio 2005 简介

Microsoft Visual Studio 2005 是 Microsoft 推出的新一代集成开发环境，包含了许多强大的工具，支持多种编程语言(C#、VB、C++、HTML 与 JavaScript 等)。VC++ 2005 是 Visual Studio 2005 开发套件之一，具有集成开发环境，可提供编辑 C、C++，以及 C++/CLI 等编程语言。

VC++ 2005 包括许多完全集成的工具，设计这些工具的目的是使编写 C++程序的整个过程更加轻松。作为 IDE 组成部分提供的 VC++ 2005 的基本部件有编辑器、编译器、连接器和库。这些是编写和执行 VC++ 2005 程序所必需的基本工具。这些工具的功能如下。

(1) 编辑器

编辑器给用户提供创建和编辑 VC++ 2005 源代码的交互式环境。除了那些肯定已经为人所熟知的常见功能(如剪切和粘贴)之外，编辑器还用颜色来区分不同的语言元素。编辑器能够自动识别 C++语言中的基本单词，并根据其类别给它们分配某种颜色。这不仅有助于使代码的可读性更好，而且在输入这些单词时出错的情况下可以提供清楚的指示。

(2) 编译器

编译器将源代码转换为目标代码，并检测和报告编译过程中的错误。编译器可以检测各种因无效或不可识别的程序代码引起的错误，还可以检测诸如部分程序永远不能执行这样的结构性错误。编译器输出的目标代码存储在称做目标文件的文件中。编译器产生的目标代码有两种类型。这些目标代码通常使用以.obj 为扩展名的名称。

(3) 连接器

连接器组合编译器根据源代码文件生成的各种模块，从作为 VC++ 2005 组成部分提供的程序库中添加所需的代码模块，并将所有模块整合成可执行的整体。连接器也能检测并报告错误。例如，程序缺少某个组成部分，或者引用了不存在的库组件。

(4) 库

库只不过是预先编写的例程集合，它通过提供专业制作的标准代码单元，支持并扩展了C++语言。我们可以将这些代码合并到自己的程序中，以执行常见的操作。Visual C++ 2005提供了各种库包括的例程所实现的操作，从而节省了用户亲自编写并测试实现这些操作的代码所需的工作量，大大提高了生产率。

1.4 简单的 VC++ 2005 程序

1.4.1 VC++ 2005 程序的开发过程

把设计好的 VC++ 2005 程序交计算机运行，并最终获得结果，这一系列工作主要由计算机完成，我们只需做一些比较简单的操作。

(1) 编辑 VC++ 2005 程序

第一项工作是把程序作为一个文本(字符)文件输入到计算机中。编辑的工作主要包括创建新程序文件(一般是.cpp 文件)，输入，浏览，修改，插入，删除等。

任何一种文本编辑器都可以完成这项工作，大多数 C++系统如 Visual C++、Borland C++等都提供了集成开发环境 IDE(Integrated Develop Environment)。IDE 不仅为用户提供了方便的编辑功能，也包括了文件管理、编译、链接和项目管理等多方面的支持。

由于 IDE 为用户提供了编辑、编译、链接、调试等综合环境。例如，在编好一个程序后，可以当即编译、运行，再根据编译和运行情况(包括系统提供的出错信息和调试手段)修改程序，使程序迅速调试通过或修改得更好。

(2) 编译和链接过程

VC++ 2005 程序不能直接交计算机运行，需经编译系统(Compiler)编译，变成由机器指令组成的可执行程序，然后由计算机运行。

编译前的 VC++ 2005 程序称为源程序，编译后的机器语言程序称为目标程序，如图 1-3 所示。

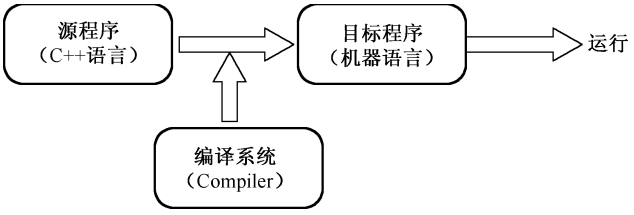


图 1-3 源程序编译为目标程序

实际上从 VC++ 2005 源程序到可执行的机器语言程序还有两个步骤必须完成:

① 编译预处理。在源程序被编译之前，须经“预处理”，其任务是根据程序中的预处理指令，完成指定的预处理任务。本节给出的简单程序中包含预处理指令：

```
#include <iostream.h>,
using namespace std;
```

其具体操作是在系统提供的 std 名字空间的标准库中找出 iostream 头文件，并把它嵌入到该预处理指令的位置。

② 链接(linking)。链接的任务是把已编译好的目标程序与其他需共同运行的目标程序和系统提供的库程序链接起来，形成可执行程序。

其中，VC++ 2005 源程序为.cpp 文件，目标程序为.obj 文件，可执行程序为.exe 文件。在调试中根据编译过程、链接过程、运行过程和结果中发现的出错信息，对源程序进行反复修改，最终得到正确的程序和结果。

目前 VC++ 2005 语言的实现系统(编译系统)很多，版本也很多，但上述过程是基本一致的，都按下面的方式处理：

(1) 一般用户编写的 VC++ 2005 程序可以按.cpp 文件的形式保存。一个.cpp 文件可以包括一个或多个程序单元。一个程序单元可以存为一个.cpp 文件，也可分存在.h 和.cpp 两个文件中。有的 C++系统的保存形式不是.cpp 文件，而是.cc 文件。

(2) 编译预处理不产生中间文件，预处理后自动进入编译，编译后的目标文件一般为.obj 文件。

(3) 经过链接处理的可执行程序记为.exe 文件(.exe 文件是可执行文件)。

(4) 运行可执行程序。

1.4.2　简单的 VC++ 2005 程序示例

下面通过“Hello，World!”这个示例程序教会大家如何编写一个简单的应用程序。

(1) 启动 Visual Studio 2005

启动 Visual Studio 2005，系统会显示如图1-4所示的集成开发环境。

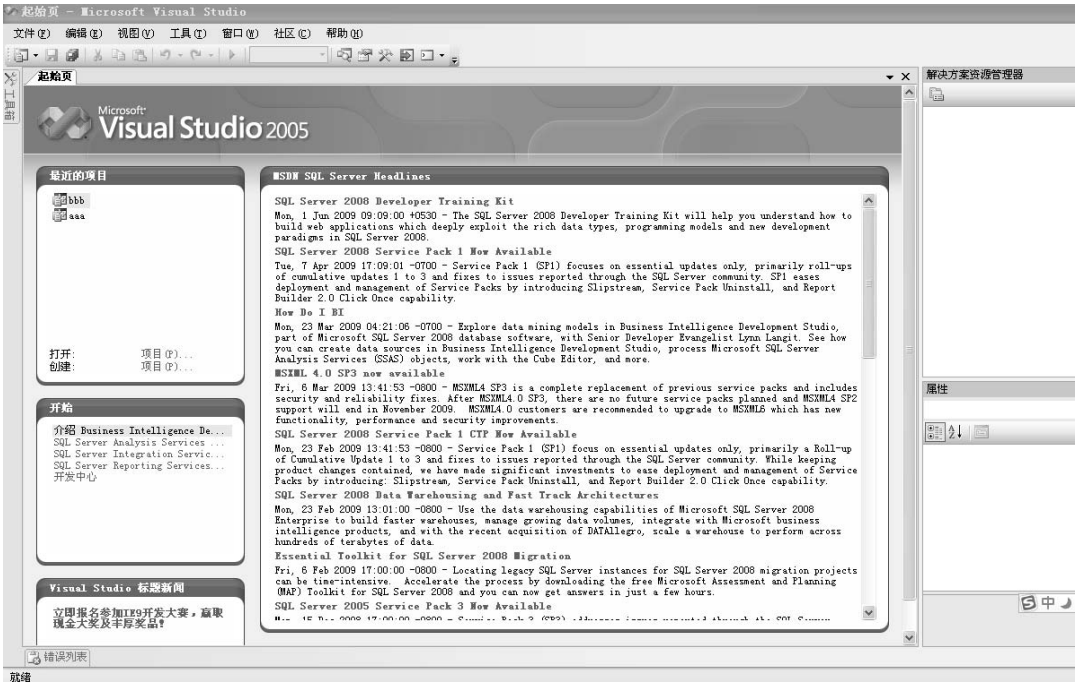


图 1-4　Visual Studio 2005 集成开发环境

在此集成开发环境中，可以使用 VC、VB、C#、ASP 开发各种应用程序。图1-4中的“最近的项目”列出了新近曾经在此环境中使用过的项目，单击某项目的名称，就可以在 IDE 中打开该项目并对它进行各种处理。图 1-4 左边是“解决方案资源管理器”窗口，它提供项目及其文件的有组织的视图，并且提供对项目和文件相关内容的便捷访问。

(2) 新建一个空项目

① 选择 Visual Studio 中的“文件”→“新建”→“项目”菜单项，系统将弹出如图1-5所示的“新建项目”对话框。

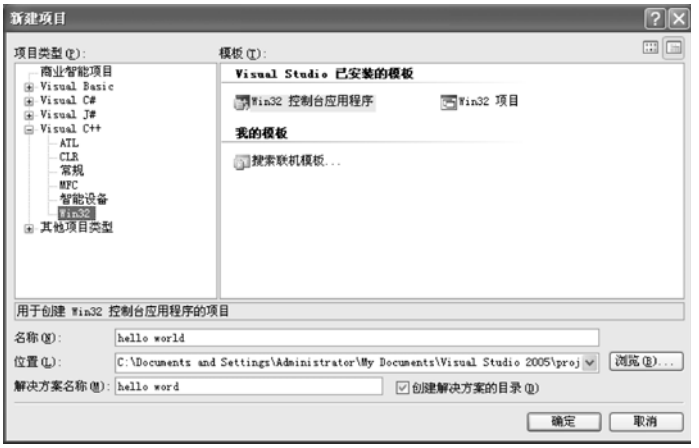


图 1-5 Visual Studio 的“新建项目”对话框

在图1-5中，列出了 Visual Studio 2005 能够使用的编程语言，能够开发的程序类型。选中窗口左边“项目类型”窗口中的 Visual VC++，并展开其树型列表，可以看出 Visual C++ 2005 支持 ATL、CLR、MFC 和智能设备等多种类型的程序设计。在右边的“模板”窗口中，列出了 Visual C++ 2005 已安装的程序模板，它可以以向导方式引导程序员建立这些应用程序的模型。

② 选择图 1-5 中的“Win32 控制台应用程序”，并在窗口下侧的“名称”文本框和“位置”下拉列表框中输入程序名和保存它的磁盘目录。在本例中，输入程序名 hello world，保存目录默认。

③ 输入项目名称和保存位置并单击“确定”按钮后，系统会显示如图1-6所示的应用程序向导对话框。

图 1-6 VC++ 2005 应用程序向导对话框

④ 选中图1-6中的“应用程序类型”选项区域中的“控制台应用程序”单选按钮，选中“附加选项”选项区域中的“空项目”复选框。单击“完成”按钮后，系统将根据前面的设

置在默认目录下创建一个空的 Win32 控制台应用程序项目，并显示如图 1-7 所示的程序设计界面。此时发现“解决方案资源管理器”窗口中有了内容，包括头文件、源文件和资源文件三部分。由于前面的操作建立的是一个空项目，所以这三个目录中并无内容。但前面的向导过程已为 hello world 程序建立了编译运行的必备架构：解决方案和项目文件。现在只需要在此架构中添加正确的源程序，它就能够被编译执行。打开保存本项目的目录，可以发现其中有个 hello world 目录，此目录中包括 hello world.sln 和 hello world.ncb 两个文件，还有一个内层目录 hello world。

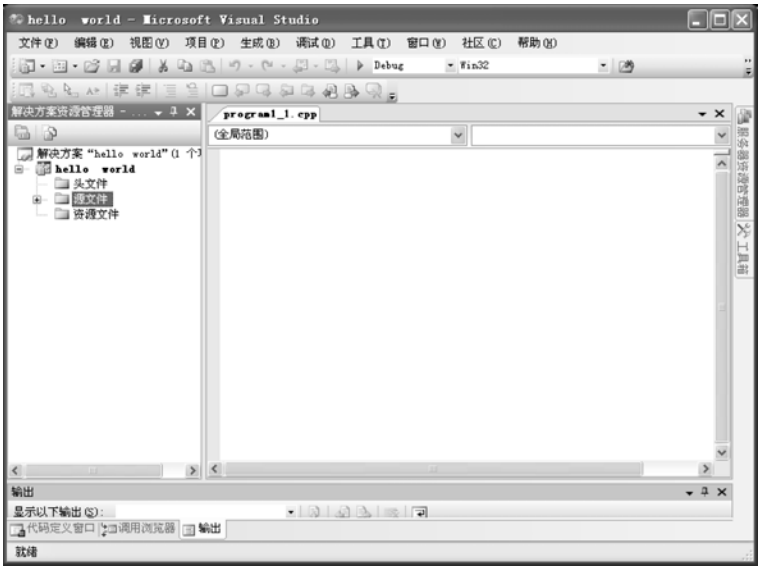


图 1-7　Visual C++ 2005 程序设计界面

图 1-7 所示的文件夹称为解决方案文件夹。解决方案是 Visual Studio 为了使集成开发环境能够应用它的各种工具、设计器、模板和设置而实现的一种概念上的容器，是用于管理 Visual Studio 配置、生成和部署相关项目集的方式。一个解决方案可以只包含一个项目，也可以包含由开发小组联合生成的多个项目。

Visual Studio 提供了解决方案文件夹，用于将相关项目组织为组，然后对这些项目组执行操作。解决方案中的每个项目可以包含多个文件或项，项目中所包含的项的类型会依据创建它们时所使用的开发语言而有所变化。总的说来，解决方案文件夹中常包括以下内容：

① 一个扩展名为.sin 的文件，记录了解决方案中的项目信息。

② 一个扩展名为.suo 的文件，记录了用户应用到的解决方案的选项信息。

③ 一个扩展名为.ncb 的文件，记录了解决方案准备的智能感知信息。智能感知是一种在编辑窗口中输入程序源码时的自动提示功能，可用于校证输入数据的正确性，还可用它简化数据输入。解决方案的项目文件夹，若一个解决方案由多个项目构成，则它会为每个项目建立一个文件夹，并将一个项目的全部内容保存在其中。一个解决方案至少包含一个项目文件夹。创建有多项目的解决方案时，在默认情况下创建的第一个项目成为启动项目。

项目是 Visual Studio 组织程序的一种容器，它包含了程序的所有相关内容，一个项目可能是一个控制台程序、一个 Web 程序或一个窗体应用程序。一个项目可能由多个文档组成，包括源程序文件、头文件及各种辅助文件。项目的所有文件都存储在项目文件夹中，图 1-7

就是空项目 hello world 的项目文件夹。随着项目中文件的增加，会有越来越多的文件添加到该文件夹中。当该项目被编译后，Visual Studio 还会在其中创建一个 Debug 目录，并将该项目编译链接过程中产生的输出结果保存在 Debug 目录中。概言之，项目文件夹中常包括以下内容：

① 一个扩展名为.vcproj 的 XML 文件，记录了本项目的详细信息。

② 本项目相关的头文件和源程序文件。

③ 一个 Debug 目录，其中保存在本项目编译和链接过程中产生的各种文件，如.exe（程序可执行文件）、.obj（源程序被编译后的目标文件，是源程序的机器码形式）、.pch（预编译头文件，用于减少重新预编译的时间）、.ilk（保存项目链接信息，用于重新编译项目时供编译器使用，避免每次修改代码时都重新链接全部代码）、.pdb（保存程序运行调试模式时的调试信息）和.idb（保存重新生成解决方案的信息）。

（3）在 hello world 项目中添加源程序

建立了程序的解决方案和项目后，可以编辑、修改、调试项目中的程序代码，还可以在解决方案中添加新项目，也可以在已有项目中添加各种程序文件。前面的向导过程建立了 hello world 程序的解决方案和项目，但所建立的是一个空项目，没有程序源文件。现在向该项目添加源文件 hello world. cpp，步骤如下。

① 用鼠标右键单击 Visual Studio 集成环境中“解决方案管理器”窗口中的 hello world 项目名，或其中“源文件”的文件夹（见图 1-7），从弹出的快捷菜单中选择“添加”→“新建项”命令，系统将弹出如图1-8所示的“添加新项”对话框。

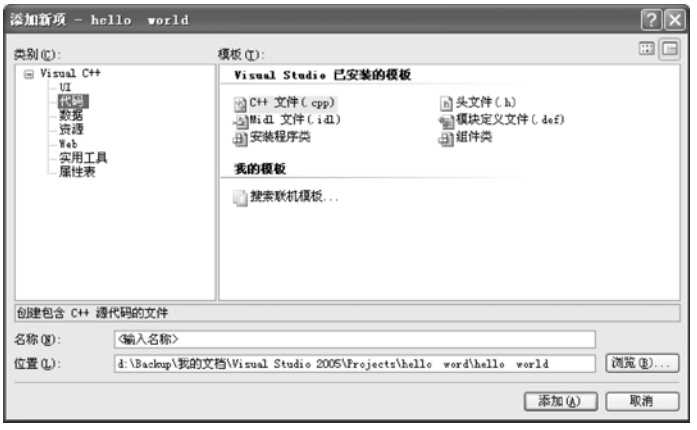


图 1-8 “添加新项”对话框

② 选中“模板”列表框中的“C++文件(.cpp)”项目，在下面的“名称”文本框中输入源文件的名称 hello world.cpp。然后单击“添加”按钮，Visual Studio 将把源文件 hello world.cpp 添加到 hello world 的项目文件夹中，并在“解决方案管理器”窗口中 hello world 项目的源文件目录中显示该文件的名字。

③ 双击“源文件”目录图标下的 hello world.cpp 文件名，然后在右边的程序编辑窗口中输入 hello world. cpp 文件的下述源代码。

【例 1.1】 一个简单的控制台程序。

```
#include< iostream>
using namespace std
```

```
    int main()
{
cout<<"Hello, World! "<< endl;
return 0;
}
```

在上述源代码中，cout 是一个输出语句，它将运算符“<<”后面的内容输出到显示屏幕上；endl 是回车换行的意思。cout 和 endl 都是在 iostream 头文件中定义的，所以在程序中用#include 将此头文件包含到程序中来。

(4) 执行程序

源代码输入完成后，选择“生成”→“生成解决方案”命令，Visual Studio 就会进行程序编译，如果没有错误就会在项目文件夹中产生 hello world. exe 命令程序，该程序可在控制台直接执行。

按 Ctrl + F5 键执行本程序，将在屏幕上看到输出结果：Hello, World!

本 章 小 结

本章主要讲述计算机程序设计语言的发展过程，并简单介绍了 VC++ 2005 语言的特点，以及 Visual C++ 2005 集成开发环境的使用。

编程人员想要得到正确且易于理解的程序，必须采用良好的程序设计方法。结构化程序设计和面向对象的程序设计是两种主要的程序设计方法。结构化程序设计建立在程序的结构定理基础之上，主张只采用顺序、选择和循环三种基本的程序结构和自顶向下逐步求精的设计方法，实现单入口单出口的结构化程序；面向对象的程序设计主张按人们通常的思维方式建立问题区域的模型，设计尽可能自然的表现客观世界和求解方法的软件，对象、消息、类和方法是实现这一目标而引入的基本概念，面向对象程序设计的基本点在于对象的封装性和继承性，以及由此带来的实体的多态性。与结构化程序设计相比较，面向对象的程序设计具有更多的优点，适合开发大规模的软件工程项目。

C++语言是当今最流行的高级程序设计语言之一，它既支持结构化的程序设计方法，也支持面向对象的程序设计方法。使用 Microsoft Visual C++ 2005 提供的集成开发环境，编程者可以轻松完成 C++项目的创建、编译、调试和运行。

习　题　1

一、选择题

1．最初的计算机编程语言是：

A．机器语言　　B．汇编语言　　C．高级语言　　D．低级语言

2．结构化程序设计的基本结构不包含以下哪项?

A．顺序　　B．分支　　C．跳转　　D．循环

3．下列哪项不是面向对象程序设计的主要特征?

A．封装　　B．继承　　C．多态　　D．结构

二、填空题

1．汇编程序的功能是将汇编语言所编写的源程序翻译成由_____组成的目标程序。

2．目前，有两种重要的程序设计方法，分别是_____和_____。

3．在 C++中，封装是通过_____来实现的。

三、问答题

1．简述高级程序设计语言相对于低级语言的优点。

2．面向对象程序设计的基本思想是什么？什么是对象、消息和类？什么是面向对象程序设计的基本特征？

3．C++语言具有哪些特点？

第 2 章　VC++ 2005 程序设计基础

2.1　VC++ 2005 基本语法

2.1.1　字符集

字符集是构成 C++语言的基本元素。用 C++语言编写程序时，除字符型数据外，其他所有成分都只能由字符集中的字符构成。C++语言的字符集由下述字符构成：

大小写的英文字母：A～Z，a～z。

数字字符：0～9。

特殊字符：空格　！　#　%　^　&　*　_(下画线)　+　=　-　～　<　>　/　\　'　"　;　.　,　()　[]　{}。

2.1.2　词法记号

词法记号是最小的词法单元，下面介绍 C++的关键字、标识符、字面常量、运算符和分隔符。

1．关键字

关键字(key word)是一类有特定的专门含义的单词。对于 C++语言来说，凡是列入关键字表的单词，一律不得移做它用。因此，关键字又称为保留字(reserved word)。如 int、for、if 等单词就属于关键字。

例如，for 是一个关键字，它在 C++程序中常常出现，必须用在 for 语句(一种循环语句)的开头。换句话说，在 C++程序中，关键字 for 指明，在它后面的应是一个 for 语句，关键字 for 只有这样一种用法。

再如，const 是另一个关键字，它用在常量说明的开头，指出在它后面说明的是常量。不过关键字 const 在 C++程序中的用法不唯一，例如，const 还可以出现在函数的参数表中，可以用 const 指明某一引用型参数是不被改变的等，所以关键字 const 有多于一种的含义和用法。

关于 C++语言的关键字，有如下说明：

(1) C++语言的关键字包含了几乎所有的 C 语言的关键字。

(2) 随着 C++语言的不断完善，其关键字集也在不断变化。例如，在由 Stroustrup 提出的原始版本的 C++语言中，尚不包括 private、protected 等关键字，至于像 template 等涉及模板概念的关键字，更是在最近的 C++版本中才增加的。

(3) 各不同版本 C++语言的实现可能有不少涉及其应用领域的关键字的设置，如与微机有关的 far、near 等关键字。还有一些个别关键字，虽然包含于两种不同的版本之中，却有不同的含义，如 volatile 关键字的使用。

总之，关键字集合是使用 C++语言编程前应首先弄清楚的，特别是对少数个别的关键字

的设置应有所了解，以免在编程中产生错误，至少应避免在设定标识符时与关键字重名。

在本节中不可能介绍所有关键字的作用，这些关键字将陆续出现在以后的章节中，逐渐使读者准确地了解所有关键字的意义和用法。

表 2-1 关键字表

asm	auto	bool	break
case	catch	char	class
const	continue	default	delete
do	double	elae	enum
extern	false	float	for
friend	goto	if	inline
int	long	namespace	new
operator	private	protected	public
register	return	short	signed
sizeof	static	struct	switch
template	this	throw	true
try	typedef	union	unsigned
using	virtual	void	volatile
wchar_t	while		

2．标识符

标识符(identifier)是由程序员为程序中的各种成分：变量、有名常量、用户定义的类型、枚举类型的值、函数及其参数、类、对象等所起的名字。名字不能随便起，必须符合标识符的组成规则：

(1) 标识符是一个以字母或下画线‘_’开头的，由字母、数字、下画线组成的字符串，如 abcd、c5、_PERSON_H 都是合法的标识符，而 3A、A*B、$ 43.5A 都是不合法的。标识符中间不可插入空格。

(2) 标识符应与任一关键字有区别，如 for、if、case 等都不可做标识符。

(3) 标识符中字母区分大小写，即 Abc 与 abc 被认为是不同的两个标识符。与此相反，关键字不区分大小写，如 FOR、For、for、foR 都认为是同一关键字，且都不可作为标识符。

(4) 标识符的有效长度。如果程序中的标识符过长，系统将对有效长度之外的字符忽略不计，一般 C++语言设有效长度为 32。

除了符合规则之外，为了在大型程序中区分和记忆，用户在为常量、变量、函数等起名字时，往往不是简单地用 a、b、c、n1、n2 等这样的名字，而是使标识符有一定的描述性，表示母鸡数量的变量名为 hen，表示我的年纪的变量名为 myage、my_age 或 myAge；还有一种“匈牙利标记法”，在变量的名字中，不但要表示其含义，还要表示数据的类型，如 imyAge 表示整型变量，ipmyAge 表示整型指针变量。

3．字面常量

C++程序中的常量是指固定不变的量。常量有两种表示形式：一种称为有名常量，一种称为字面常量(literal constant)。如圆周率 pai=3.1416，其中 pai 就是个有名常量，pai 是量 3.1416 的名字，而 3.1416 称为字面常量。

C++程序中有名常量的名字就是一个标识符，而字面常量是一类特殊的单词，它是程序

所要处理的数据的值。字面常量分为 4 类：int 型常量，float 型常量，char 型常量和字符串常量。关于字面常量将在 2.2.2 节详细叙述。

4．运算符

运算符(operator)主要由字母、数字之外的第三类基本符号组成，个别关键字如 sizeof、new、delete，也被列入运算符之列，其余运算符为：

+，–，* ，/，%，==，!= ，<，<=，>，>=，!，&&，‖ ，&，^ ，|,~，++ ，– – ，+=，–=，*= ，/=，%=，<<=，>>=，&= ，^= ，|= ，?:，=，()，[]，. ，–> ，<<，>>,’，:

关于运算符的分类、功能和用法将在后续章节中分别介绍，这里仅做几点说明。

(1) C++语言与大多数常见的高级语言相比，是运算符和运算形式最为丰富的一种语言，学习 C++语言应努力掌握其运算符的用法，特别是包含混合操作的运算符的用法。例如，前后缀增量减量运算符++、– –的功能和用法，复合赋值运算符+=、–=的功能和用法。一般情况下，正确使用这些运算符，可使程序简明、清晰。

(2) 在上面所列的运算符中，许多运算符身兼二任或兼多任，如运算符*：

对于整数 m、n，m*n 表示整数乘法；

对于实数 a、b，a*b 表示浮点数乘法；

对于某类指针 p，*p 表示 p 指向变量的内容。

此外，*除了做运算符之外，还有另外的功能，例如：

```
int * i，j;
```

这里的*已不是运算符，可以说它起到一个关键字或分隔符的作用。由此可知，C++语言中单词的分类不是绝对的。

(3) 运算符的概念和用法来自数学，不过在 C++语言中运算符和运算的概念对于人们日常习惯的理解已有所扩展。例如，在 C++语言中，赋值(“=”)、表达式的并列(<表达式>,<表达式>)、下标(a[i]中的[ ])都一律作为运算和运算符来处理，对此读者应予以注意。

5．分隔符

分隔符(separator)本身没有明确的含义，但程序中却必不可少，一般用来界定或分割其他语法成分。程序中的分隔符有点像文章中的标点符号。例如：

;：表示一个语句的结束。

"：表示一个字符串的开始与结束。

分隔符包括：

␣(空格)　"　#　(，)　/*　*/　//　’　;　{　}

其中较为特殊的是空格，程序中空格的使用十分重要。C++程序允许连续的空格出现，从语法功能看，连续多个空格与一个空格作用相同，在两个相邻的关键字或标识符之间起到了分割的作用。另外，在程序中连续的空格可改善程序的格式，提高程序的可读性。

分隔符/*、*/和//用于注释。/*与*/应成对地出现，其间的任何符号序列，在程序中等价于一个空格，这就为程序员在程序中加入注释以提高可读性提供了方便。分隔符//有类似的功能，它使其后直至行尾的任何字符序列成为注释。

另外，分隔符“,”可用来分割并列的变量、对象、参数等，同时“,”也是运算符。

2.2 基本数据类型和表达式

2.2.1 基本数据类型

C++将数据分为若干类型，程序中使用到的所有数据都必须指明其类型。定义数据类型实际上给出了两方面信息：一是该类型数据在内存中占有多大存储空间，二是该类数据能进行哪些合法运算。这种对数据的分类称为数据类型。VC++ 2005 中的数据类型分为两大类：基本数据类型和非基本数据类型。基本数据类型是 C++内部预定义的，包括字符型(char)、整型(int)、实型(float)、双精度型(double)、逻辑型(bool)和无值型(void)。非基本数据类型是用户根据程序需要，按 C++语法规则参与构造出来的数据类型，包括数组(array)、指针(pointer)、引用(reference)、类(class)、结构体(struct)、联合(union)和枚举(enum)等。C++数据类型如图2-1 所示。

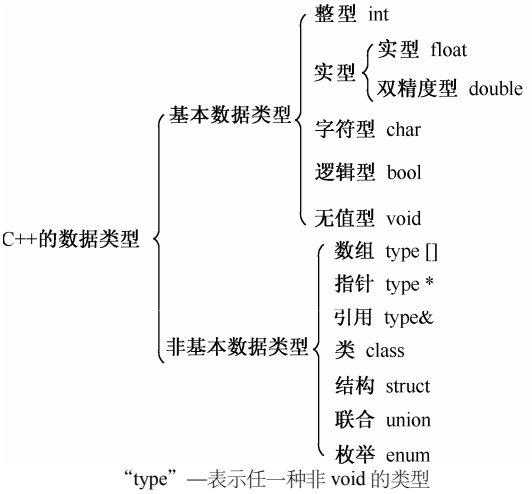


图 2-1 C++的数据类型

非基本数据类型可细分为:完全由用户定义，包括结构体(struct)、联合(union)和类(class)；部分用户定义，枚举(enum)类型；由其他类型导出，包括数组(array)、指针(pointer)和引用(reference)。

C++对基本数据类型也分别进行了封装，称为内置数据类型，内置数据类型不仅定义了数据类型，还定义了常用操作。本节仅介绍各种基本数据类型的定义，常用操作将在后面介绍。

整型用来存放整数，整数(有符号的整数)在内存中存放的是它的补码，无符号数没有符号位，存放的就是原码。整数占用的字节数与机型有关，一般占用 2 或 4 字节，32 位机上占用 4 字节。

字符型用来保存字符，存储的是该字符的 ASCII 码，占用 1 字节。如大写字母 A 的 ASCII 码为 65，在对应的字节中存放的就是 65。西文 ASCII 码表参见附录 A。字符型数据从本质上说也是整数，可以是任何一个 8 位二进制整数。

通常所说的字符型指单字符型，C++同时也支持宽字符类型(wchar_t)，或称双字节字符型。

由于汉语系字符很多，用 ASCII 字符集处理远远不够，因此又创立了双字节字符集(DBCS:double-byte character set)，每个字符用 2 字节来编码。为便于软件的国际化，国际上

一些知名公司联合制定了新的宽字节字符标准——Unicode。该标准中所有字符都是双字节的，不同的语言和字符集分别占用其中一段代码。这种用统一编码处理西文、中文及其他语言符号，就是 Unicode 码。

为了支持 Unicode，ANSI C 的头文件 string.h 中定义了名为 wchar_t 的数据类型，用来存放 Unicode 码，同时在库函数中定义了相应的处理 Unicode 的串处理函数。

宽字符类型不属于基本数据类型，限于篇幅，这里不做具体介绍，可参阅有关书籍资料。

实型和双精度型都用来存放实数，两者表示的实数精度不同。实数在内存中以规范化的浮点数存放，包括尾数、数符和阶码。数的精度取决于尾数的位数，32 位机上实型为 23 位(因规范化数的数码最高位恒为 1，不必存储，实际为 24 位)，双精度为 52 位。

逻辑型也称布尔型，其取值为 true(逻辑真)和 false(逻辑假)，存储字节数在不同编译系统中可能有所不同，VC++ 6.0 中为 1 字节。布尔型在运算中可以与整型相互转化，false 对应为 0，true 对应为 1 或非 0。

无值型主要用来说明函数的返回值类型，将在第 6 章具体介绍。

基本数据类型还可以加上一些修饰词，包括：signed(有符号)、unsigned(无符号)、long(长)、short(短)。表 2-2 为 VC++ 2005 中的所有基本数据类型。

表 2-2　VC++ 2005 中所有基本数据类型

类　型	名　称	占用字节数	取 值 范 围
bool	布尔型		true, false
(signed) char	有符号字符型	1	–128～127
unsiged char	无符号字符型	1	0～255
(signed) short (int)	有符号短整型	2	–32768～32767
unsigned short (int)	无符号短整型	2	0～65535
(signed) int	有符号整型	4	-2^{31}～$(2^{31}-1)$
unsigned (int)	无符号整型	4	0～$(2^{32}-1)$
(signed) long (int)	有符号长整型	4	-2^{31}～$(2^{31}-1)$
unsigned long (int)	无符号长整型	4	0～$(2^{32}-1)$
float	实型	4	-10^{38}～10^{38}
double	双精度型	8	-10^{308}～10^{308}
long double	长双精度型	8	-10^{4932}～10^{4932}
void	无值型	0	无值

表中用()括起来部分书写时可以省略。int 被 4 个修饰词修饰时均可以省略，另外，无修饰词的 int 和 char 默认为有符号的，等同于加修饰词 signed。

C++为强类型语言，所有数据的使用严格遵从“先说明后使用”的原则，以便编译器进行编译。在程序设计中，数据类型的误用是常见错误，这要求语言的编译器能检查出尽可能多的数据类型方面的错误。一个编译器能检查出的错误越多，则该编译器越好。

2.2.2　字面常量

字面常量指程序中直接给出的量，其值在程序执行过程中保持不变。字面常量存储在程序区，而不是数据区，对它的访问不是通过数据地址进行的。

根据取值和表示方法的不同，字面常量分为整型常量、实型常量、字符型常量和字符串常量。

1. 整型常量

整型常量即整数，在C++中可以用十进制、八进制、十六进制表示。

十进制表示与我们熟悉的书写方式相同。如：

```
15   -24
```

八进制表示以0打头，由数字0～7组成，用来表示一个八进制数。如:

```
012              //八进制数12，即十进制数10
-0655            //八进制数-655，即十进制数-429
```

十六进制以0X(大小写均可)打头，由数字0～9和字母A～F(大小写均可)组成，用来表示一个十六进制数。以下是一些常整数的例子：

```
0x32A            //十六进制数32A，即十进制数810
-0x2fe0          //十六进制数-2fe0，即十进制数-12256
```

整数常量还可以表示长整数和无符号整数。长整型常数以L或l结尾，无符号常整数以U或u结尾，以UL或LU(大小写均可)结尾则可表示无符号长整型常数。例如:

```
-84L             //十进制长整数-84
026U             //八进制表示的无符号整数26
0X32LU           //十六进制表示的无符号长整型数32
```

对于没有标明为长整型和无符号整型的整数，编译系统会根据数据大小自动识别。

2. 实型常量

C++中，包含小数点或10的幂的数为实型常量。有一般形式和指数形式两种表示方法。

一般形式与平时书写形式相同，由数字0～9和小数点组成。例如：

```
0.23    -125.76    0.0    .46    -35.
```

指数形式(也称科学表示法)表示为尾数乘以10的整数次方形式，由尾数、E或e和阶数组成。例如：

```
123E12           //表示123×10^12
-.34e-2          //表示-0.34×10^-2
```

指数形式要求在E或e前面的尾数部分必须有数字，后面的指数部分必须为整数。以下表示方法都是不合法的：

```
E4               //不能没有尾数
1.43E3.5         //阶数不能是实数
```

3. 字符型常量

字符常量是用单引号引起来的单个字符。在内存中保存的是字符的ASCII码值。在所有字符中，有些是可显示字符，通常用单引号引起来表示。如：

```
'a'              //字符a
'@'              //字符@
```

```
'4'              //字符 4
' '              //空格字符
```

除此之外，还有一些不可显示的，以及无法从键盘输入的字符，如回车符、换行符、制表符等。为了能够在程序中表示这些不可显示字符，以及其他一些特殊字符常量，VC++ 2005 提供了一种称为转义序列字符的表示方法。表 2-3 中列出了 VC++ 2005 中定义的转义序列字符及其含义。

表 2-3　VC++ 2005 中预定义的转义序列字符及其含义

字符表示	ASCII 码值	名　　称	含义或用途
\a	0x07	响铃	用于输出
\b	0x08	退格(Backspace 键)	退回一个字符
\f	0x0c	换页	用于输出
\n	0x0a	换行符	用于输出
\r	0x0d	回车符	用于输出
\t	0x09	水平制表符(Tab 键)	用于输出
\v	0x0b	纵向制表符	用于制表
\0	0x00	空字符	用于字符串结束标志等
\\	0x5c	反斜杠字符	用于需要反斜杠字符的地方
\'	0x27	单引号字符	用于需要单引号的地方
\"	0x22	双引号字符	用于需要双引号的地方
\nnn	八进制表示		用八进制 ASCII 码表示字符
\xnn	十六进制表示		用十六进制 ASCII 码表示字符

表中有三个特殊字符，本身既是字符，又在 C++中有特殊含义（“\”用来表示转义字符，“'”用来表示字符常量，“"”用来表示字符串常量），所以当它们作为字符常量出现时，不能像其他可见字符那样表示，而必须采用转义序列字符。下面是用转义序列字符表示的不可见字符和特殊字符：

```
'\n'             //换行符
'\a'             //响铃
'\\'             //反斜杠符号
```

表中最后两行是所有字符的通用表示方法，即用反斜杠加 ASCII 码表示。如字母 a 可以有三种表示方法：a、\141 和\x61，可以看出，对于可见字符，第一种是最简单直观的表示方法。

4．字符串常量

用双引号引起来的若干字符称为字符串常量。例如：

```
"I am a Chinese. "    "123"    "a"    "    "
```

字符串常量在内存中是按顺序逐个存储串中字符的 ASCII 码，并在最后存放一个“\0”字符作为串结束符。字符串的长度指的是串中“\0”之前的所有字符数量，包括不可见字符。因此，字符串常量占用的字节数是串长+1。一个字符用单引号引起来是字符常量，而用双引号引起来就是字符串常量，二者在内存中的值是不一样的。例如：

```
"a"              //占 2 字节，存放'a'和'\0'，值为 0x6100
'a'              //占 1 字节，存放'a'，值为 0x61
```

需要说明的是，单引号作为字符串中的一个字符时，可以直接按书写形式出现，也可以用转义序列字符表示；但双引号作为字符串中的一个字符时，只能用转义序列字符表示。例如：

```
"I've finished. "   //表示字符串 I've finished
"sister\'s toy. "   //表示字符串 sister's toy
"\"Book\""          //表示字符串 "Book"
```

2.2.3 变量

在程序的执行过程中其值可以变化的量称为变量，变量是需要用名字来标识的。程序编译运行时，每个变量占用一定的字节单元，并且变量名和单元地址之间存在一个映射关系，当引用一个变量时，计算机通过变量名寻址，从而访问其中的数据。

变量有类型之分，如整型变量、浮点型变量、字符变量等，还有各种导出类型变量如数组变量。对任一变量，编译程序按其类型分配一段连续的存储单元，用以保存变量的取值。

变量是数据在程序中出现的主要形式，在变量第一次被使用前应被说明。变量说明的格式为：

```
[<存储类>]<类型名或类型定义> <变量名表>;
```

例如：

```
int size,high,temp = 37;
static long sum;
auto float t = 0.5;
```

存储类：程序员可有五种选择:

auto：把变量说明为自动变量。

register：把变量说明为寄存器变量。

static：把变量说明为静态变量。

extern：把变量说明为外部变量。

默认：按自动变量处理。[<存储类>] 的方括号表示可以默认。

有关存储类的说明稍后还有详细介绍。

类型名或类型定义：任何变量说明语句中，必须包含数据类型的说明，不可默认。在上面的例中，int、long(long int)、float 就是变量的类型说明。对于有些用户定义的类型，也可以直接把类型定义本身作为变量的类型说明。例如：

```
enum color {RED = 1, YELLOW, BLUE} c1 = BLUE, c2;
```

变量名表：列出该说明语句所定义的同一类型的变量及其初值，其格式为：

```
变量名表：<变量名>[=<表达式>],<变量名表>
```

例如：

```
float c1;
char ch1 = 'e',ch2;
int a, b, c = 5;
```

(1) auto 变量：用关键字 auto 说明的局部变量，称为自动变量。该变量在程序的临时工作区中获得存储空间，如说明语句未赋初值，系统不会自动为其赋初值，随着变量生存期结束，这段临时空间将被释放，可能为其他自动变量占用。变量的 auto 属性为默认属性，即不写 auto 与写上的效果相同。

(2) register 变量：用 register 说明的局部变量，称为寄存器变量，该变量将可能以寄存器作为存储空间。register 说明仅能建议(而不是强制)系统使用寄存器，这是因为寄存器虽存取速度快，但空间有限，当寄存器不够用时，该变量仍按自动变量处理。

一般在短时间内被频繁访问的变量置于寄存器中可提高效率。不过本书并不建议经常使用 register 变量，理由如下：

① 在许多情况下使用寄存器变量效果不明显。

② 有的版本的 C++编译系统具有对局部变量按某种策略自动决定可否占用可用寄存器的功能，效果比程序员决定可能好一些。

③ 局部变量存于寄存器时它将没有内存地址，可能影响与寻址有关的操作，如寻址运算符&的操作。因此有的版本 C++语言使用关键字 volatile 来专门说明“非寄存器变量”只可占用内存。

(3) static 变量：用 static 说明的变量称为静态变量，任何静态变量的生存期将延续到整个程序的终止。其要点为:

① 静态变量和全局变量一样，在内存数据区分配空间，在整个程序运行过程中不再释放。

② 静态变量如未赋初值，系统将自动为其赋默认初值 0(NULL)。

③ 静态变量的说明语句在程序执行过程中多次运行或多次被同样说明时，其第一次称为定义性说明，进行内存分配和赋初值操作，在以后的重复说明时仅维持原状，不再做赋初值的操作。

例如：

```
static float r, s = 2.5;
```

static 变量在程序设计中常被用到，由于 static 变量是“永久”占用空间，可以保存函数调用过程中某些局部量的结果，并把它传送到该函数的下次调用之中。

由于静态变量容易浪费空间，所以不宜过多使用。

(4) extern 变量：用关键字 extern 说明的变量称为外部变量。

一个变量被说明为外部变量，其含义是告诉系统不必为其按一般变量那样分配内存，该变量已在这一局部的外面定义。

外部变量一般用于由多个文件组成的程序中，有些变量在多个文件中被说明，但却是同一变量，指出某一变量为外部变量就避免了重复分配内存，产生错误。

关于全局变量、局部变量等概念将在第 6 章介绍。

C++语法为在变量的说明语句中进行变量初始化提供了方便。除了基本类型及其派生类型的变量初始化比较简单，已在前面介绍之外，对于数组、结构、指针等类型的变量初始化也较易实现。

数组变量的初始化只须用“{”和“}”把初始值表括起来即可，例如：

```
int a[4] = {1,2,3,4},b[2][3]={1,2,2,3,3,4};
int c[3] = {6,9},d[2][2] = {{2,4},{6,8}};
```

一般把值列出即可，亦可部分元素赋初值，系统按顺序为前面的元素赋初值。高维数组的初值也可用“{”、“}”再分组。

结构变量的初始化与数组类似，应注意结构元素的类型和顺序，例如：

```
struct st{int n,m;float a; };
st stl = {2,3,4.0}, st2 = {0,0,1.0};
```

指针变量的初始化情况有所不同，一般需要取地址的运算，例如:

```
int a[4],b;
int * pa = a+2, * pb = &b;
```

指针变量 pa 的初值为数组元素 a[2]的地址，指针变量 pb 的初值为变量 b 的地址。

注：有关数组、结构、指针及引用类型的说明和使用将在后面详细介绍。

2.2.4 符号常量

在 VC++ 2005 程序中，所出现的常量通常使用符号常量来表示。符号常量就是使用一个标识符来表示某个常量值。使用符号常量不仅可增加程序的可读性，而且为修改常量值带来极大的方便。符号常量和变量一样在程序中必须遵循“先声明，后使用”的原则，程序中出现的所有符号常量都必须在使用前由常量说明语句说明。

在 C++语言中，定义常量使用类型说明符 const。具体定义格式如下：

```
const <类型说明符> <常量名> = <常量值>;
```

例如：`const int N = 2000;`

```
const float pai = 3.1416;
```

必须以关键字 const 开头。

类型名：限定为基本类型(int，float，char，bool)及其派生类型。

常量名：标识符。

表达式：其值应与该常量类型一致的表达式(常量和变量也是表达式)。

由于常量和变量同样要求系统为其分配内存单元，所以可以把字符常量视为一种不允许赋值改变的或只读不写的变量，称其为 const 变量。

C++语言另外还从 C 语言中继承了一种定义常量的方法，即在编译预处理命令中的宏定义(或宏替换)方法，例如:

```
# define N 1000
# define pai 3.1416
```

也能起到类似的作用，不过，用宏替换的方法定义符号常量与 const 方式的实现机制是不同的：宏替换是在编译时把程序中出现的所有标识符 N 或 pai 都用 1000 和 3.1416 来替换，这里并没有一个只读不写的 const 变量存在；宏替换的方式中没有类型、值的概念，仅是两个字符串的代换，容易产生问题。因此，在大多数情况下建议使用 const 常量。

2.2.5 运算符与表达式

1．运算符

(1) 算数运算符

在 C++中对常量或变量进行运算或处理的符号称为运算符，参与运算的对象称为操作数。C++的运算符非常丰富，本节只介绍包括算术运算符、关系运算符、逻辑运算符、位运算符等在内的基本运算符及其优先级和结合性。表 2-4 列出 C++中提供的运算符及其优先级。

表 2-4　VC++ 2005 的运算符及其优先级

优　先　级	运　算　符	功　　能	用　　法	结　合　性
1	::	全局域	name	左→右
		类域	class::name	
		名字空间域	namespace::name	
2	.	成员访问	object.member	左→右
	->	成员访问	pointer->member	左→右
	（）	括号	（expr）	左→右
		函数调用	name（expr_list）	
		类型构造	type（expr_list）	
	[ ]	数组下标	variable[expr]	左→右
	++	后置递增	variable++	左→右
	--	后置递减	variable--	左→右
3	typeid	类型 ID	typeid（type）	右→左
		运行时类型 ID	typeid（expr）	
	const_cast	类型转换	const_cast<type>（expr）	右→左
	dynamic_cast	类型转换	dynamic_cast<type>（expr）	右→左
	reinterpret_cast	类型转换	reinterpret_cast<type>（expr）	右→左
	static_cast	类型转换	static_cast<type>（expr）	右→左
	sizeof	对象大小	sizeof expr	右→左
		类型大小	sizeof （type）	
	（）	强制类型转换	（type） expr	右→左
	++	前置递增	++variable	右→左
	--	前置递减	--variable	右→左
	~	按位取反	~expr	右→左
	!	逻辑非	!expr	右→左
	+	单目正	+expr	右→左
	-	单目负	-expr	右→左
	*	间接引用	*pointer	右→左
	&	取地址	&variable	右→左
	new	分配对象	new type	右→左
		分配并初始化对象	new type（expr_list）	
		分配数组	new type []	
	delete	释放对象	delete pointer	右→左
		释放数组	delete [] pointer	
4	->*	间接访问指针指向的类成员	pointer->*pointer_to_member	左→右
	.*	访问指针指向的类成员	object.*pointer_to_member	左→右
5	*	乘	expr*expr	左→右
	/	除	expr/expr	左→右
	%	求余（取模）	expr%expr	左→右
6	+	加	expr+expr	左→右
	-	减	expr-expr	左→右

（续表）

优 先 级	运 算 符	功 能	用 法	结 合 性
7	<<	按位左移	expr<<expr	左→右
	>>	按位右移	expr>>expr	左→右
8	< <= > >=	比较大小	expr operator expr	左→右
9	== !=	比较是否相等	expr operator expr	左→右
10	&	按位与	expr & expr	左→右
11	^	按位异或	expr ^ expr	左→右
12	|	按位或	expr | expr	左→右
13	&&	逻辑与	expr && expr	左→右
14	||	逻辑或	expr || expr	左→右
15	?:	条件运算	expr?expr:expr	右→左
16	=	赋值	variable=expr	右→左
	+= -= *= /= %= <<= >>= &= |= ^=	复合赋值	variable operator expr	右→左
17	throw	抛出异常	thow expr	右→左
18	,	逗号	expr , expr	左→右

优先级和结合性决定了运算中的优先关系。运算符的优先级指不同运算符在运算中的优先关系，表中序号越小，优先级越高。运算符的结合性决定同优先级的运算符对操作数的运算次序。若一个运算符对其操作数按从左到右的顺序运算，称该运算符为右结合，反之称为左结合。如计算 10+20，对运算符“+”，是先取 10，再取 20，然后做加法运算，即按从左到右的顺序执行运算，所以 运算符“+”是右结合的。再如 a+=35，对运算符“+=”，是先取 35，再取变量 a，做加法运算后将结果赋值给变量 a，即按从右向左的顺序运算，所以运算符“+=”是左结合的。

按照要求的操作数个数，运算符分为单目（一元）运算符、双目（二元）运算符和三目（三元）运算符。单目运算符只对一个操作数运算，如负号运算符“–”等；双目运算符要求有两个操作数，如乘号运算符“*”等；三目运算符要求有三个操作数，三元运算符只有一个“?:”。

按照运算的种类，运算符可分为算术运算符、关系运算符、逻辑运算符、位运算符等。表 2-5 介绍了常用运算符的优先级。

表 2-5 算术运算符及优先级

优 先 级	运 算 符	名 称
3	+	正，单目
	-	负，单目
5	*	乘，双目
	/	除，双目
	%	求余，双目
6	+	加，双目
	-	减，双目

对于乘法运算符“*”和除法运算符“/”，当两个操作数均为整数时，结果为整数，除法运算后舍去小数取整；只要有一个操作数是实数，结果就是实数。例如：

```
5/4              //结果为 1，整数
5/4.0            //结果为 1.25，实数
```

求余运算符“%”也称求模运算符，要求两个操作数均必须是整数，结果为两个整数相除后的余数。如果两个整数中有负数，则先用两数绝对值求余，最后结果的符号与被除数相同。例如：

```
6%3              //结果为 0
6%7              //结果为 6
7%6              //结果为 1
-7%6             //结果为-1
7%-6             //结果为 1
-7%-6            //结果为-1
```

C++中算术运算应注意数据溢出问题，即运算结果超出对应数据类型的表示范围。编译程序只会对除法运算时除数为 0 这种情况提示出错，而整数的加、减和乘法运算产生溢出的情况，系统不作为错误处理，程序将继续执行并产生错误的计算结果。因此，程序设计者必须在程序中检查并处理整数溢出问题。

(2) 关系运算符和逻辑运算符

关系运算符都是二元运算符，包括>(大于)、>=(不小于)、<(小于)、<=(不大于)、==(等于)和!=(不等于)。

关系运算符完成两个操作数的比较，结果为逻辑值 true(真)或 false(假)。在 C++中这两个逻辑值与整数之间有一个对应关系，真对应 1，假对应 0；反过来，0 对应假，非 0 整数对应真。所以关系运算结果可以作为整数参与算术运算、关系运算、逻辑运算及其他运算。

逻辑运算符用来进行逻辑运算。其操作数和运算结果均为逻辑量。运算结果同样可以作为一个整数参与其他运算。关于逻辑运算符的种类及语义见表 2-6。

表 2-6　逻辑运算符

优 先 级	运 算 符	名　称	语　义
3	!	逻辑非，单目	操作数的值为真，则结果为假
12	&&	逻辑与，双目	当两个操作数全为真时，结果为真，否则为假
13	||	逻辑或，双目	两个操作数中有一个为真，则结果为真

由于逻辑值和整数之间的对应关系，也允许整型操作数进行逻辑运算：

```
21&&0            //逻辑与，21 与 0，结果为假:0
21||0            //逻辑或，21 或 0，结果为真:1
!21              //逻辑非，21 的非，结果为假:0
```

(3) 位运算符

C++语言提供位运算，这是其他高级语言不具备的。它对操作数的各位进行操作。位运算符共有 6 个：~(按位取反)、<<(左移)、>>(右移)、&(按位与)、|(按位或)、^(按位异或)。其中按位取反为单目运算符，其余为双目运算符。

① 按位取反运算符“~”。将操作数的每个二进制位取反，即 1 变为 0，0 变为 1。例如，整数 a 的值为 10011011，则~a 的值为 01100100。

② 左移运算符“<<”。运算一般格式为：

```
a<<n
```

其中 a 为整数，n 为一个正整数常数。语义为将 a 的二进制数依次向左移动 n 个二进制位，并在低位补 0。移位运算不影响 a 本身的值，而是只产生一个中间量，这个中间量被引用后即不再存在。例如，变量 a 的值为 00000010，则 a<<3 的值为 00010000，而 a 的值仍为 00000010。

整数左移 1 位相当于该数乘以 2，左移 n 位相当于乘以 2n，但移位运算的速度比乘法快。由于左移等同于乘法，因此也可能出现溢出。

③ 右移运算符“>>”。与左移运算符类同，将左操作数向右移动右操作数指定的二进制位数，忽略移位后的小数部分，并在高位补 0。一个整数右移 n 位相当于除以 2n，但比除法快。例如，变量 a 的值为 00010000，则 a>>2 的值为 00000100，a 的值仍为 00010000。

在 C++中有符号数右移时高位补符号位，如：

```
-32>>3      //-32 右移 3 位，由 11110000B 得 11111110B，结果为-2
```

④ 按位与运算符“&”。将两个操作数的对应位逐一进行按位逻辑与运算。运算规则为：对应位均为 1 时，该位运算结果为 1；否则为 0。例如，整型变量 a 和 b 的二进制值分别为 01001101 和 00001111，则 a&b 的值为 00001101，用竖式更容易看出结果：

a	01001101
b	00001111
a & b	00001101

该运算可用来将整数的某些位置 0，而保留所需要的位，上例保留了低 4 位。

⑤ 按位或运算符“|”。将两个操作数的对应位逐一进行按位逻辑或运算。运算规则为：只要有一个数对应位为 1，该位运算结果即为 1；两个数对应位均为 0，该位结果为 0。例如，对上例的整数 a、b，a | b 的值为 01001111，用竖式表示为：

a	01001101
b	00001111
a | b	01001111

该运算符可用来将整数的某些位置 1。上例高 4 位不变，低 4 位全 1。

⑥ 按位异或运算符“^”。将两个操作数的对应位逐一进行按位异或运算。运算规则为：当对应位的值不同时，该位运算结果为 1，否则为 0。例如，对上例的整数 a、b, a^b 的值为 01000010，用竖式表示为：

a	01001101
b	00001111
a ^ b	01000010

该运算符可用来将一个整数的某些位取反，或将整型变量的值置 0(将整型变量与自身按位异或)。上例低 4 位取反，高 4 位不变。

需要说明的一点是，以上例子中的整数都只取了低 8 位。

(4) 赋值运算符

对程序中的任何一种数据的使用包括赋值和引用。将数据存放到相应存储单元中称为赋值，如果该单元中已有值，赋值操作以新值取代旧值；从某个存储单元中取出数据使用，称为引用，引用不影响单元中的值，即一个量可以多次引用。常量只能引用，不能赋值。

赋值通过赋值运算符“=”来完成，其意义是将赋值号右边的值送到左边变量所对应的单元中。赋值号不是等号，它具有方向性。C++将变量名代表的单元称为“左值”，而将变量的值称为“右值”。左值必须是内存中一个可以访问且可以合法修改的对象，因此只能是变量名，而不能是常量或表达式，关于表达式的概念稍后介绍。例如，下面的赋值运算是错误的：

```
3.1415926=pi          //左值不能是常数
x+y=z                 //左值不能是表达式
const int N=30;
N=40                  //左值不能是常变量
```

(5) 自增、自减运算符

C++提供了两个具有给变量赋值作用的单目算术运算符：自增运算符“++”和自减运算符“--”，其意义是使变量当前值加 1 或减 1，再赋给该变量。例如:

```
i++                   //相当于 i=i+1
j--                   //相当于 j=j-1
```

由于具有赋值功能，这两个运算符要求操作数只能是变量，不能是常量或表达式。运算符的使用还分前置和后置两种，上例是运算符后置。当自增、自减的变量还参与其他运算时，运算符前置和后置的结果一般是不同的。前置是先增减后引用，即先对变量自加或自减，用新的值参与其他运算；后置则是先引用后增减，即用变量原来的值参与其他运算，然后再对变量进行自加或自减。例如：

```
int i=5, j=5, m, n;
m=i++;                //相当于 m=i; i=i+1; 结果 i 的值为 6，m 的值为 5
n=++j;                //相当于 j=j+1;n=j; 结果 j 的值为 6，n 的值为 6
```

(6) sizeof()运算符

该运算符用于计算操作数类型或变量的字节数。一般格式为:

```
sizeof (数据类型)  或  sizeof(变量名)
```

其中数据类型可以是标准数据类型，也可以是用户自定义类型。变量必须是已定义的变量。括号可以省略，运算符与操作数之间用空格间隔。例如:

```
sizeof(int)           //值为 4
sizeof float          //值为 4
double x;
sizeof x              //值为 8
```

使用该运算符是为了实现程序的可移植性和通用性，因为同一操作数类型在不同计算机上可能占用不同字节数。

其他运算符将在后续章节中陆续介绍。

2．表达式

由运算符、操作数及标点符号组成的，能取得一个值的式子称为表达式。表达式中的操作数可以是常量、变量或函数等。一个常数或变量即是最简单的表达式。表达式的求值要根据运算符的意义、求值次序、优先级、结合性，以及类型转换约定进行。根据运算符的不同，表达式有算术表达式、关系表达式、逻辑表达式、赋值表达式、逗号表达式等。

(1) 算术表达式

由算术运算符连接的表达式称为算术表达式。例如：

```
a+5*b
(x+y) % n
```

使用算术表达式还应注意以下三点：

① 两个运算分量应为同一类型，如果不同，应该遵循类型转换原则，由“短”类型向“长”类型的自动转换，即按照：char→int→float→double 的次序进行自动转换。例如：

```
int a,b;
float x,y;
x=b*a+y;
```

表达式中 a、b 和 y 虽然不是同一类型，a*b 的结果是 int 型，它相对于 y 的 float 类型是“短”类型，于是 a*b 的结果转化为 float 型和 y 相加，然后赋值给 x。

② 两个 int 型数据相除，结果应为 int 型，若商不是整数，也要取整。int 型与 float、double 型相除，结果应为 float 或 double 型。例如：

```
int a = 3, b=2;
float y = 2.0;
```

a/b 的值是 1 而不是 1.5，而 a/y 的值是 1.5。这是因为 a/b 的结果是 int 型，所以取 1。而 a/y 中 y 是 float 型，结果应该是 float 型，所以取 1.5。

③ 取模运算符%主要应用于整形数值计算。a%b 表示用 b 除 a 所得到的余数。例如，47%4 的值为 3，33%19 的值为 14。因此对于整数(int 和 char 型)来说，除法运算和取模运算有如下关系：

```
a- b*(a /b)= a%b //这里“=”为等号
```

(2) 关系表达式

由关系运算符连接的表达式称为关系表达式。关系表达式的值为 true 或 false。这个值可对应整数 1 或 0 直接参与其他运算。例如：

```
a>b>c           //等同于(a>b)>c，先求 a>b 的值，再将结果 0 或 1 与 c 比较大小
a+b>c+d         //等同于(a+b)>(c+d)，结果为 0 或 1
```

(3) 逻辑表达式和逻辑表达式求值的优化

由逻辑运算符连接的表达式称为逻辑表达式。逻辑表达式的值为 true 或 false。这个值可对应整数 1 或 0 参与其他运算。例如，求下列逻辑表达式的值：

```
int a=0, b=2, c=3;
```

```
float x=1.8, y=2.4;
a>b&&a<c|| (x>y)-!a
```

根据优先级，该表达式等同于

```
((a>b)&&(a<c))|| ((x>y)-!a)
```

求值顺序为：先求 a>b，值为 0；再求出 a<c，值为 1；再求 0&&1，值为 0；求 x>y，值为 0；再求!a，值为 1；再求 0−1，值为−1，作为逻辑值为 1；最后求 0||1，值为 1。所以整个表达式的值为 1。

对于较为复杂的逻辑表达式，建议使用配对括号，以省去记忆并减少出错的可能性。

(4) 赋值表达式与复合赋值表达式

赋值表达式的格式为：

〈变量〉=〈表达式〉

赋值表达式的含义是，先计算右边表达式的值，再将该值赋给左边的变量。赋值表达式本身也取得值，左值就是赋值表达式的值。请看以下几例:

```
a=5+6              //合法，计算 5+6，将 11 赋给 a，整个表达式的值为 11
d=c=b=a+1          //合法，计算 a+1，将 12 赋给 b，再将表达式(b=a+1)的值 12 赋给 c，
                   //再将表达式(c=b=a+1)的值 12 赋给 d，整个表达式的值为 12
c=(a=1)+(b=2)      //合法，将 1 赋给 a，再将 2 赋给 b，再将表达式(a=1)和(b=2)的
                   //值相加得 3，将 3 赋给 c，整个表达式的值为 3
a=3+b=2            //非法，因为算术运算符的优先级高，先计算表达式 3+b，该表
                   //达式成为第二个赋值号的左值
```

在 C++中，所有的双目算术运算符和位运算符均可与赋值运算符组合成一个单一运算符，称为复合运赋值算符。包括以下 10 个：

+= −= *= /= %= <<= >>= &= |= ^=

复合赋值运算符的格式与赋值运算符完全相同，表示为：

变量　复合赋值运算符　表达式

它等同于

变量 = 变量　运算符 表达式

例如：

```
a+=1               // a=a+1
a*=b-c             // a=a*(b-c)
a-=(b+1)           // a=a-(b+1)
```

复合赋值运算表达式仍属于赋值表达式，它不仅简化书写，而且能提高表达式的求值速度。

(5) 逗号表达式

用逗号连接起来的表达式称为逗号表达式。一般格式为:

表达式 1,表达式 2,…,表达式 *n*

它所做的运算是，从左到右依次求出各表达式的值，并将最后一个表达式的值作为整个逗号表达式的值。逗号运算符的优先级最低。例如，假定 a=1，b=2，c=3，逗号表达式：

```
a=a+1, b=b*c, c=a+b+c
```

运算过程是，将 2 赋给 a，将 6 赋给 b，将 2+6+3 即 11 赋给 c，并将 11 作为整个逗号表达式的值。再如，以下三个表达式的结果是不同的：

```
c=b=(a=3, 4*3)          //结果为:a=3, b=12, c=12, 表达式的值为 12
c=b=a=3, 4*3            //结果为:a=3, b=3, c=3, 表达式的值为 12
c=(b=a=3, 4*3)          //结果为:a=3, b=3, c=12, 表达式的值为 12
```

并非所有的逗号都构成逗号表达式，有些情况下逗号只作为分隔符，如函数的参数之间用逗号分隔：

```
max(a+b, c+d)
```

2.2.6 语句

语句是程序的基本单位。C++中的语句分为以下几种。

1. 表达式语句

表达式语句是最简单的语句形式，在表达式后面加上一个分号就构成了表达式语句，一般格式为：

```
表达式;
```

例如，赋值表达式可以构成赋值语句。

2. 空语句

只由一个分号构成的语句称为空语句。空语句不执行任何操作，但具有语法作用，如 for 循环在有些情况下循环体是空语句，也有些情况下循环条件判别是空语句，这些将在第 3 章的循环语句中介绍。大多数情况下，从程序结构的紧凑性与合理性角度考虑，尽量不要随便使用空语句。

3. 复合语句

由一对“{}”括起来的一组语句构成一个复合语句。复合语句描述一个块，在语法上起一个语句的作用。

对单个语句，必须以“;”结束；对复合语句，其中的每个语句仍以“;”结束，而整个复合语句的结束符为“}”。

4. 流程控制语句

流程控制语句用来控制或改变程序的执行方向。具体内容在第 3 章流程控制语句中介绍。

2.3 数据的输入与输出

2.3.1 I/O流

在 C++中，将数据从一个对象到另一个对象的流动抽象为“流”。流在使用前要被建立，使用后要被删除。从流中获取数据的操作称为提取操作，向流中添加数据的操作称为插入操作。数据的输入与输出是通过 I/O 流来实现的，cin 和 cout 是预定义的流类对象。cin 用来处理标准输入，即键盘输入；cout 用来处理标准输出，即屏幕输出。

2.3.2　预定义的插入符和提取符

“<<”是预定义的插入符，作用在流类对象 cout 上便可以实现最一般的屏幕输出。

格式如下：

```
cout<<表达式<<表达式…
```

在输入语句中，可以串联多个插入运算符，输出多个数据项。在插入运算符后面可以写任意复杂的表达式，系统会自动计算出它们的值并传递给插入符。例如：

```
cout<<"Hello! \n";
```

将字符串“Hello!”输出到屏幕上并换行。

```
cout<<"a+b="<<a+b;
```

将字符串“a+b=”和表达式 a+b 的计算结果依次在屏幕上输出。

最一般的键盘输入是将提取符作用在流类对象 cin 上。格式如下：

```
cin>>表达式>>表达式...
```

在输入语句中，提取符可以连续写多个。每个后面跟一个表达式，该表达式通常是用于存放输入值的变量。例如：

```
int a,b;
cin>>a>>b;
```

要求从键盘上输入两个 int 型数，两数之间以空格分隔。若输入：

```
3　4
```

这时，变量 a 得到的值为 3，变量 b 得到的值为 4。

2.3.3　简单的I/O格式控制

当使用 cin 和 cout 进行数据的输入/输出时，无论处理的是什么类型的数据，都能够自动按照正确的默认格式处理。但这还是不够，我们经常需要设置特殊的格式。设置格式有很多方法，将在后续章节详细介绍，本节只介绍最简单的格式控制。

C++ I/O 流类库提供了一些操纵符，可以直接嵌入到输入/输出语句中来实现 I/O 格式控制。使用操纵符，首先必须在源程序的开头包含 iomanip 头文件。表2-7 中列出了几个常用的 I/O 流类库操纵符。

表 2-7　常用的 I/O 流类库操纵符

操 纵 符 名	含　义
Dec	数值数据采用十进制表示
Hex	数值数据采用十六进制表示
Oct	数值数据采用八进制表示
Ws	提取空白符
Endl	插入换行符，并刷新流
Ends	插入空字符
setprecision (int)	设置浮点数的小数位数(包括小数点)
setw (int)	设置域宽

例如，要输出浮点数 3.14159 并换行，设置域宽为 6 个字符，小数点后保留 3 位有效数字，输出语句如下:

```
cout<<setw(6)<<setprecision(4)<<3.14159<<endl;
```

2.4 基于VC++ 2005 的简单程序开发

2.4.1 一个简单程序设计例程

下面通过一个非常简单的 C++程序，用来了解 C++程序的组成，以及建立程序的过程。这里不详细介绍所有的细节，因为这些内容将在后面章节中探讨。

【例 2.1】 一个简单的 C++程序。

```
#include <iostream>
using namespace std;
int main ( )
{
   cout<<"欢迎学习 Visual Studio C++2005! ";
   return 0;
}
```

程序编译、运行后计算机屏幕会显示“欢迎学习 Visual Studio C++ 2005!”字样。

该程序由一个函数 main()组成。函数是代码的一个自包含块，用一个名称表示，在本例中是 main。程序中还可以有许多其他代码，但每个 C++程序至少要包含函数 main()，且只能有一个 main()函数。C++程序的执行总是从 main()的第一条语句开始的。

该函数的第一行语句是：

```
int main( )
```

这行语句指出，这是函数 main 的开始。开头的 int 表示函数在执行完后返回一个整数值。因为是函数 main()，所以最初调用它的操作系统会接收这个值。

函数 main()包含两个可执行语句，每个语句放在一行上：

```
cout<<"欢迎学习 Visual Studio C++2005! ";
return 0;
```

这两个语句会按顺序执行。通常情况下，函数中的语句总是按顺序执行，除非有一个语句改变了执行顺序。第 3 章将介绍什么类型的语句可以改变执行顺序。

在 C++中，输入和输出是使用流来执行的。如果要从程序中输出消息，可以把该消息放在输出流中；如果要输入消息，则把它放在输入流中。因此，流是数据源或数据池的一种抽象表示。在程序执行时，每个流都关联着某个设备，关联着数据源的流就是输入流，关联着数据目的地的就是输出流。对数据源或数据池使用抽象表示的优点是，无论流代表什么，编程都是相同的。例如，从磁盘文件中读取数据的方式与从键盘上读取完全相同。在 C++中，标准的输出流和输入流称为 cout 和 cin，在默认情况下，它们分别对应计算机屏幕和键盘。

main()中的第一行代码利用插入运算符“<<”把字符串“欢迎学习 Visual Studio C++ 2005!”放在输出流中，从而把它输出到屏幕上。在编写涉及输入的程序时，应使用提取运算符>>。

函数体中的第二个语句，也是最后一个语句：

```
return 0;
```

结束了该程序，把控制权返回给操作系统。它还把值 0 返回给操作系统，也可以返回其他值来表示程序的不同结束条件，操作系统还可以利用该值来判断程序是否执行成功。一般情况下，0 表示程序正常结束，非 0 值表示程序不正常结束。但是，非 0 返回值是否起作用取决于操作系统。

#include <iostream>中的 iostream 是头文件。头文件中的代码定义了一组可以在需要时包含在程序源文件中的标准功能。C++标准库中提供的功能存储在头文件中，但头文件不仅仅用于这个目的。我们可以创建自己的头文件，包含自己的代码。在这个程序中，cout 在头文件 iostream 中定义。这是一个标准的头文件，它提供了在 C++中使用标准输入和输出功能所需要的定义。如果程序不包含下面的代码行:

```
#include <iostream>
```

就不会进行编译，因为<iostream>头文件包含了 cout 的定义，没有它，编译器就不知道 cout 是什么。这是一个预处理指令，详见本节后面的内容。#include 的作用是把<iostream>头文件的内容插入程序源文件中该指令所在的位置。这是在程序编译之前完成的。

在尖括号和标准头文件名之间没有空格。在许多编译器中，两个尖括号“<”和“>”之间的空格是很重要的，如果在这里插入了空格，程序就可能不编译。

“using namespace std;”这行语句告诉编译器，使用 C++自带的标准命名空间里面的东西，那个 std 就是标准的命名空间。

命名空间这个概念不难理解，因为 C++程序功能强大，它不但自己内置了一套非常完善的程序库来供大家编程用之外，还允许程序员开发自己的库。这就存在一个问题，因为本身自带的库太多太复杂，我们不可能都记住，那么自己定义的库里面如果命令的名字刚好跟自带的重复了，那么编译器就会不知道是选择自带的还是自己写的命令了。为了解决这个问题，C++就采用了命名空间，它可以避免大家在使用自带库和自定义库的时候可能出现的重复错误。比如这个例子里面我们用了标准库里面的 iostream 的 cout 指令，在程序开始我们用“using namesapce std;”来告诉编译器，我们用的是系统自带的标准库里面的东西，而不是我们自己写的 cout 指令。

命名空间的定义方法是“using namespace 命名空间名;”，这里我们用了标准空间 std 的内容，所以就定义“using namespace std;”这样的语句。using namespace 是一个完整的语句，所以注意结尾是有分号的。

下面将着重对 main 函数、注释、编译预处理、命名空间等概念做详细介绍。

2.4.2　main函数

函数是 C++语言中最重要的概念之一，有关函数的设计还将在第 6 章详细介绍，主函数也是函数，还有一些细节将在那里介绍。

以 main 命名的主函数是 C++程序中具有特殊性质和功能的函数。在 C++程序中，main 函数是整个程序的入口。

(1) 主函数是任何一个 C++程序中唯一必不可少的函数。

(2) 主函数的函数名是标识符 main，它是由系统指定的。

(3) 主函数的类型(返回类型)为 int 型，它可以返回一整型数值，这个值是传送给操作系统的。

(4) 主函数的参数可有下面的两种形式：

```
int main( )
int main( int argc, char *argv[] )
```

即主函数的形参表也是由系统规定的，用户不得随意设计。

(5) 主函数可调用任何其他函数，但它本身不可由任何函数调用，而只能由 C++程序编译后的执行代码在其上运行的操作系统自动调用，实际上它是为系统运行该程序标识出启动地址。因此，主函数也是任一 C++程序运行的执行入口。

(6) 主函数不可做其他属性说明，如不可说明为静态(static)、内联(inline)等。

总之，C++程序的主函数是：

① 整个程序的主控模块。

② 程序的入口。

③ 程序和它的运行环境的接口。

主函数以返回值和参数的方法提供了程序和它的运行环境之间交换信息的手段。

2.4.3 注释

注释是用来帮助程序员读程序的语言结构，可以用来概括程序的算法、标识变量的意义，或者阐明一段比较难懂的程序代码。注释不会增加程序的可执行代码的长度。在代码生成以前，编译器会将注释从程序中剔除掉。

C++中有两种注释符号，一种是注释对“/* */”。注释的开始用“/*”标记，编译器会把“/*”与“*/”之间的代码当做注释。注释可以放在程序的任意位置，可以含有制表符 tab、空格或换行，还可以跨越多行程序。第二种注释符是双斜线“//”，它可用来注释一个单行，注释符右边的内容都将被当做注释而被编译器忽略。例如：

```
/*代码作用在屏幕上输出  Hello World!  */
#include<iostream> //需要包含的头文件
using namespace std;
void main( )
{
    cout << "Hello World!";
}
```

2.4.4 编译预处理

可以在 C++源程序中加入一些“预处理命令”(preprocessor directives)，以改进程序设计环境，提高编程效率。预处理命令是 C++统一规定的，但是它不是 C++语言本身的组成部分，不能直接对它们进行编译(因为编译程序不能识别它们)。

现在使用的 VC++ 2005 编译系统包括了预处理、编译和连接等部分，因此不少用户误认为预处理命令是 C++语言的一部分，甚至以为它们是 C++语句，这是不对的。必须正确区别预处理命令和 C++语句，区别预处理和编译，才能正确使用预处理命令。C++与其他高级语言的一个重要区别是可以使用预处理命令和具有预处理的功能。

VC++ 2005 提供的预处理功能主要有：宏定义，文件包含，条件编译。分别用宏定义命

令、文件包含命令、条件编译命令来实现。为了与一般 C++语句相区别，这些命令以符号“#”开头，而且末尾不包含分号。

1．宏定义

可以用#define 命令将一个指定的标识符(即宏名)来代表一个字符串。定义宏的作用一般是用一个短的名字代表一个长的字符串。它的一般形式为：

```
#define 标识符 字符串
```

这就是已经介绍过的定义符号常量。例如：

```
#define PI 3.1415926
```

还可以用#define 命令定义带参数的宏定义。其定义的一般形式为：

```
#define 宏名(参数表) 字符串
```

例如：

```
#define S(a,b) a*b            //定义宏 S(矩形面积)，a、b 为宏的参数
```

使用的形式如下：

```
area=S(3,2)
```

用 3、2 分别代替宏定义中的形式参数 a 和 b，即用 3*2 代替 S(3, 2)。因此赋值语句展开为：

```
area=3*2;
```

由于 C++增加了内置函数(inline)，比用带参数的宏定义更方便，因此在 C++中基本上已不再用#define 命令定义宏了，主要用于条件编译。

2．“文件包含”处理

(1)“文件包含”的作用

所谓“文件包含”处理是指一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。C++提供了#include 命令用来实现“文件包含”的操作。如在程序中有以下#include 命令：

```
#include "file2.cpp"
```

它的作用如图2-2所示。

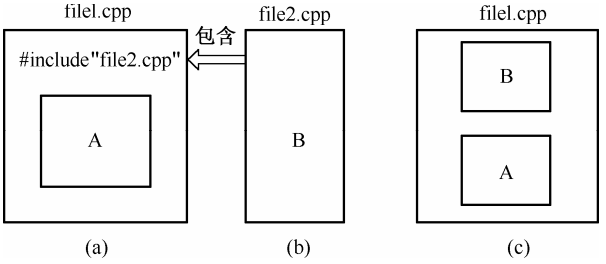


图 2-2　文件包含示意图

“文件包含”命令是很有用的，它可以节省程序设计人员的重复劳动。

#include 命令的应用很广泛，绝大多数 C++程序中都包括#include 命令。现在，库函数的

开发者把这些信息写在一个文件中，用户只需将该文件“包含”进来即可(如调用数学函数的应包含 cmath 文件)，这就大大简化了程序，写一行#include 命令的作用相当于写几十行、几百行甚至更多行的内容。这种常用在文件头部的被包含的文件称为“标题文件”或“头部文件”。

头文件一般包含以下几类内容：

① 对类型的声明。

② 函数声明。

③ 内置(inline)函数的定义。

④ 宏定义，用#define 定义的符号常量和用 const 声明的常变量。

⑤ 全局变量定义。

⑥ 外部变量声明，如 extern int a;。

⑦ 还可以根据需要包含其他头文件。

不同的头文件包括以上不同的信息，提供给程序设计者使用，这样程序设计者不需自己重复书写这些信息，只需用一行#include 命令就把这些信息包含到本文件了，大大地提高了编程效率。由于有了#include 命令，就把不同的文件组合在一起，形成一个文件。因此说，头文件是源文件之间的接口。

(2) include 命令的两种形式

在#include 命令中，文件名除了可以用尖括号括起来以外，还可以用双撇号括起来。#include 命令的一般形式为：

```
#include <文件名>
```

或

```
#include "文件名"
```

如：

```
#include <iostream>
```

或

```
#include "iostream"
```

都是合法的。二者的区别是：用尖括号时，系统到系统目录中寻找要包含的文件，如果找不到，编译系统就给出出错信息。

有时被包含的文件不一定在系统目录中，这时应该用双撇号形式，在双撇号中指出文件路径和文件名。

如果在双撇号中没有给出绝对路径，如#include "file2.c"则默认指用户当前目录中的文件。系统先在用户当前目录中寻找要包含的文件，若找不到，再按标准方式查找。如果程序中要包含的是用户自己编写的文件，宜用双撇号形式。

对于系统提供的头文件，既可以用尖括号形式，也可以用双撇号形式，都能找到被包含的文件，但显然用尖括号形式更直截了当，效率更高。

(3) 关于 C++标准库

在 C++编译系统中，提供了许多系统函数和宏定义，而对函数的声明则分别存放在不同

的头文件中。如果要调用某个函数，就必须用#include 命令将有关的头文件包含进来。C++的库除了保留 C 的大部分系统函数和宏定义外，还增加了预定义的模板和类。但是不同 C++，库的内容不完全相同，由各 C++编译系统自行决定。

不久前推出的 C++标准将库的建设也纳入进来，规范化了 C++标准库，以便使 C++程序能够在不同的 C++平台上工作，便于互相移植。新的 C++标准库中的头文件一般不再包括后缀.h，例如：

```
#include <string>
#include <iostream>                    //C++形式的头文件
```

如果用户自己编写头文件，可以用.h 为后缀。

3．条件编译命令

一般情况下，在进行编译时对源程序中的每行都要编译。但有时希望程序中某一部分内容只在满足一定条件时才进行编译，也就是指定对程序中的一部分内容进行编译的条件。如果不满足这个条件，就不编译这部分内容。这就是“条件编译”。有时，希望当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。

条件编译命令常用的有以下形式：

(1)

```
#ifdef 标识符
程序段 1
#else
程序段 2
#endif
```

它的作用是当所指定的标识符已经被#define 命令定义过，则在程序编译阶段只编译程序段 1，否则编译程序段 2 。#endif 用来限定#ifdef 命令的范围。其中#else 部分也可以没有。

(2)

```
#if 表达式
程序段 1
#else
程序段 2
#endif
```

它的作用是当指定的表达式值为真(非零)时就编译程序段 1 ，否则编译程序段 2 。可以事先给定一定条件，使程序在不同的条件下执行不同的功能。

在调试程序时，常常希望输出一些所需的信息，而在调试完成后不再输出这些信息。可以在源程序中插入条件编译段。下面是一个简单的示例。

```
#include <iostream>
using namespace std;
#define RUN                                    //在调试程序时使之成为注释行
int main( )
{ int x=1,y=2,z=3;
```

```
#ifndef RUN                                    //本行为条件编译命令
cout<<"x="<< x <<",y="<< y <<",z="<< z ;       //调试程序需输出该信息
#endif                                         //本行为条件编译命令
cout<<"x*y*z="<<x*y*z<<endl;
}
```

第 3 行用#define 命令的目的不在于用 RUN 代表一个字符串，而只是表示已定义过 RUN，因此 RUN 后面写什么字符串都无所谓，甚至可以不写字符串。在调试程序时去掉第 3 行(或在行首加//，使之成为注释行)，由于无此行，故未对 RUN 定义，第 6 行据此决定编译第 7 行，运行时输出 x、y、z 的值，以便用户分析有关变量当前的值。运行程序输出：

```
x=1,y=2,z=3
x*y*z=6
```

在调试完成后，在运行之前，加上第 3 行，重新编译，由于此时 RUN 已被定义过，则该 cout 语句不被编译，因此在运行时不再输出 x, y, z 的值。运行情况为：

```
x*y*z=6
```

2.4.5 命名空间与using应用

很多初学 C++的人，对于 C++中的一些基本但又不常用的概念感到模糊，命名空间(namespace)就是这样一个概念。

C++中采用的是单一的全局变量命名空间。在这单一的空间中，如果有两个变量或函数的名字完全相同，就会出现冲突。当然，你也可以使用不同的名字，但有时我们并不知道另一个变量也使用完全相同的名字；有时为了程序的方便，必需使用同一名字。比如你定义了一个变量 string user_name，有可能在你调用的某个库文件或另外的程序代码中也定义了相同名字的变量，这就会出现冲突。命名空间就是为解决 C++中的变量、函数的命名冲突而服务的。解决的办法就是将你的变量定义在一个不同名字的命名空间中。就好像张家有电视机，李家也有同样型号的电视机，但我们能区分清楚，就是因为他们分属不同的家庭。

在 C++中，名称(name)可以是符号常量、变量、宏、函数、结构、枚举、类和对象等。为了避免在大规模程序的设计中，以及在程序员使用各种各样的 C++库时，这些标识符的命名发生冲突，标准 C++引入了关键字 namespace(命名空间)，可以更好地控制标识符的作用域。

命名空间是一种描述逻辑分组的机制，可以将按某些标准在逻辑上属于同一个集团的声明放在同一个命名空间中。命名空间可以是全局的，也可以位于另一个命名空间之中，但是不能位于类和代码块中。所以在命名空间中声明的名称(标识符)，默认具有外部链接特性(除非它引用了常量)。在所有命名空间之外，还存在一个全局命名空间，它对应于文件级的声明域。因此，在命名空间机制中，原来的全局变量现在被认为位于全局命名空间中。

标准 C++库中所包含的所有内容(包括常量、变量、结构、类和函数等)都被定义在命名空间 std(standard，标准)中了。

1．定义命名空间

有两种形式的命名空间——有名的和无名的，它们的定义方法分别为：

```
namespace 命名空间名 { // 有名命名空间
```

```
[声明序列]
}
```

或

```
namespace { // 无名命名空间
[声明序列]
}
```

命名空间的成员，是在命名空间定义中的花括号内声明了的名称。可以在命名空间的定义内定义命名空间的成员(内部定义)；也可以只在命名空间的定义内声明成员，而在命名空间的定义之外，定义命名空间的成员(外部定义)。

命名空间成员的外部定义的格式为：

```
命名空间名::成员名 ……
```

注意：不能在命名空间的定义中声明(另一个嵌套的)子命名空间，只能在命名空间的定义中定义子命名空间。也不能直接使用“命名空间名::成员名 ……”定义方式，为命名空间添加新成员，而必须先在命名空间的定义中添加新成员的声明。另外，命名空间是开放的，即可以随时把新的成员名称加入到已有的命名空间中去。方法是，多次声明和定义同一命名空间，每次添加自己的新成员和名称。例如：

```
namespace A { int i; void f(); }   // 现在 A 有成员 i 和 f()
namespace A { int j; void g(); }   // 现在 A 有成员 i、f()、j 和 g()
```

2．使用命名空间

使用命名空间的方法有三种。

(1) 作用域解析运算符(::)

对命名空间中成员的引用，需要使用命名空间的作用域解析运算符::。例如：

```
std::cout << "Hello, World!" << std::endl;
```

(2) using 指令(using namespace)

为了省去每次调用命名空间成员和标准库的函数和对象时，都要添加“命名空间名::”和“std::”的麻烦，可以使用标准 C++的 using 编译指令来简化对命名空间中的名称的使用。格式为：

```
using namespace 命名空间名[::子命名空间名……];
```

在这条语句之后，就可以直接使用该命名空间中的标识符，而不必写前面的命名空间定位部分。因为 using 指令，使所指定的整个命名空间中的所有成员都直接可用。例如：

```
using namespace std;
      cout << "Hello, World!" << endl;
```

又例如：

```
using namespace System:: Drawing::Imaging;
      using namespace System::Window::Forms::Design::Behavior;
```

(3) using 声明(using)

除了可以使用 using 编译指令(组合关键字 using namespace)外，还可以使用 using 声明来简化对命名空间中的名称的使用。格式为：

```
using 命名空间名::[命名空间名::……]成员名;
```

注意：关键字 using 后面并没有跟关键字 namespace，而且最后必须为命名空间的成员名(而在 using 编译指令的最后，必须为命名空间名)。

与 using 指令不同的是，using 声明只是把命名空间的特定成员的名称添加该声明所在的区域中，使得该成员可以不需要采用(多级)命名空间的作用域解析运算符来定位，而直接被使用。但是该命名空间的其他成员，仍然需要作用域解析运算符来定位。例如：

```
using std::cout;
      cout << "Hello, World!" << std::endl;
```

3．using指令与using声明的比较

可见，using 编译指令和 using 声明都可以简化对命名空间中名称的访问。

using 指令使用后，可以一劳永逸，对整个命名空间的所有成员都有效，非常方便。而 using 声明，则必须对命名空间的不同成员名称一个一个地去声明，非常麻烦。

但是，一般来说，使用 using 声明会更安全。因为，using 声明只导入指定的名称，如果该名称与局部名称发生冲突，编译器会报错。而 using 指令导入整个命名空间中的所有成员的名称，包括那些可能根本用不到的名称，如果其中有名称与局部名称发生冲突，则编译器并不会发出任何警告信息，而只是用局部名去自动覆盖命名空间中的同名成员。特别是命名空间的开放性，使得一个命名空间的成员，可能分散在多个地方，程序员难以准确知道，别人到底为该命名空间添加了哪些名称。

虽然使用命名空间的方法有多种可供选择，但是不能贪图方便，一味使用 using 指令，这样就完全背离了设计命名空间的初衷，也失去了命名空间应该具有的防止名称冲突的功能。

一般情况下，对偶尔使用的命名空间成员，应该使用命名空间的作用域解析运算符来直接给名称定位。而对一个大命名空间中的经常要使用的少数几个成员，提倡使用 using 声明，而不应该使用 using 编译指令。只有需要反复使用同一个命名空间的多数成员时，使用 using 编译指令，才被认为是可取的。

本 章 小 结

本章主要讲述了 VC++ 2005 语言的基本知识。在 C++语言中，语句、变量、函数、预处理指令、输入和输出等是重要的概念，应该在编程实践中逐渐掌握这些概念和它们的应用。

标识符是用来标识变量、函数、数据类型等的字符序列。C++中的标符可以由大写字母、小写字母、下画线(_)和数字 0～9 组成，但必须以大写字母、小写字母或下画线(_)开头。C++语言中预定义了一些标识符，称为关键字，它们不能被再定义。

布尔型、字符型、整型、浮点型和空类型是基本数据类型。指针、数组、引用、结构和类可以通过基本数据类型进行构造，称为复合数据类型。

变量就是机器一个内存位置的符号名，在该内存位置可以保存数据，并可通过符号名进行访问。为了提高程序的可读性，给变量命名时应该注意使用有意义的名字。变量第一次赋值称为初始化，变量在使用之前应当先声明。

常量是在程序运行过程中，其值不能改变的量。

C++语言本身没有输入/输出功能，而是通过输入/输出库完成 I/O 操作。C 程序使用的 stdio I/O（标准 I/O）库也能够在 C++中使用；另外 C++语言还提供了一种称为 iostream（I/O 流库）的 I/O 库。

本章还介绍了 C++的各种基本运算符构成（算术运算符、关系运算符、逻辑运算符、位运算符、条件运算符、赋值运算符、逗号运算符及其他运算符），以及它们的优先级和结合性，介绍了由运算符组成的各种表达式。

自增、自减运算符，前缀式是先将操作数增 1（或减 1），然后取操作数的新值参与表达式的运算。后缀是先将操作数增 1（或减 1）之前的值参与表达式的运算，到表达式的值被引用之后再做加 1（或减 1）运算。

关系运算符两边的数值结果必须是类型相同的。

在实现优先级与实际需要不相符时，需要使用括号来改变。

参加运算的两个操作数类型不同时，C++将自动做隐式类型转换，但有时要做强制类形转换。

表达式和语句的一个重要区别是：表达式具有值；而语句是没有值的，并且语句末尾要加分号。

最后以一个简单的 VC++ 2005 程序为例，介绍了 main 函数、注释、编译预处理和命名空间等概念。

习　题　2

一、选择题

1. C++程序的执行总是从哪里开始的?
   A. main 函数　　B. 第一行　　C. 头文件　　D. 函数注释
2. 字符型数据在内存中的存储形式是：
   A. 原码　　B. 补码　　C. 反码　　D. ASCII 码
3. 下面常数中不能作为常量的是：
   A. 0xA5　　B. 2.5e-2　　C. 3e2　　D. 0583
4. 以下选项中是正确的整型常量的是:
   A. 1.2　　B. –20　　C. 1,000　　D. 674
5. 以下选项中不是正确的实型常量的是:
   A. 3.8E-1　　B. 0.4e2　　C. –43.5　　D. 243.43e-2
6. 以下符号中不能作为标识符的是：
   A. _256　　B. Void　　C. Scanf　　D. Struct
7. 下面不能正确表示 a*b/(c*d)的表达式是：
   A. (a*b)/c*d　　B. a*b/(c*d)　　C. a/c/d*b　　D. a*b/c/d
8. 下列运算符中，运算对象必须是整型的是:
   A. /　　B. %=　　C. =　　D. &
9. 若 x、y、z 均被定义为整数，则下列表达式终能正确表达代数式 1/(x*y*z)的是：
   A. 1/x*y*z　　B. 1.0/(x*y*z)　　C. 1/(x*y*z)　　D. 1/x/y/(float)z
10. 已知 a、b 均被定义为 double 型，则表达式 b=1, a=b+5/2 的值为：

A. 1 B. 3 C. 3.0 D. 3.5

11. 如有 int a=11;则表达式(a++*1/3)的值是：

A. 0 B. 3 C. 4 D. 12

12. 在下列运算符中，优先级最低的是：

A. || B. != C. < D. +

13. 表达式 9!=10 的值为:

A. 非零值 B. True C. 0 D. 1

14. 能正确表示 x>=3 或者 x<1 的关系表达式是：

A. x>=3 or x<1 B. x>=3|x<1 C. x>=3||x<1 D. x>=3&&x<1

15. 下列运算符中优先级最高的是:

A. ! B. % C. -= D. &&

二、填空题

1. 若 a 为 double 型的变量，表达式 a=1, a+5, a++的值为_____。
2. 表达式 7.5+1/2+45%10 =_____。
3. 与! (x>2)等价的表达式是_____。
4. 表达式与语句的重要区别是_____。
5. 赋值运算符的结合性是由_____至_____。
6. x *= y+8 等价于 x=_____。
7. 给出下列程序的输出结果____。

```
#include <iostream>
using spacename std;
void main( )
{
int a=3,b=6;
    int c=a^b<<2;
    cout<<c<<endl;
  }
```

8. 给出下列程序的输出结果____。

```
#include <iostream>
using spacename std;
  void main()
  {
    int x=5;
    int y=2+(x+=x++,x+8,++x);
    cout<<y<<endl;
  }
```

9. 给出下列程序的输出结果____。

```
#include <iostream>
using spacename std;
  void main( )
```

```
{
  int a=7,b=4;
  float x,y=27.2,z=3.4;
  x=a/2+b*y/z+1/3;
  cout<<x<<endl;
}
```

10. 给出下列程序的输出结果____。

```
#include <iostream>
using spacename std;
void main( )
{
  int a=-1,b=4,k;
  k=(a++<=0)&&!(b--<=0);
  cout<<k<<a<<b<<endl;
}
```

三、简答题

1. x=2, y=3, z= 4 时，计算下面表达式的值 A1、A2。
   A1 = x + y +2/2+z 和 A2=x+(y+2)/(2-z)
2. 叙述算术运算符的组成。
3. 将下列运算符按优先级从高到低进行排序：‘+’、‘*’、‘&&’、‘&’、‘>’、‘>=’、‘*=’。
4. 叙述下列运算符各能代表几种意义：‘–’、‘&’、‘*’。
5. 下列运算符的结合性如何：‘+’、‘&’、‘=’、‘||’。
6. 计算下列表达式的值：
   i. 1/2 + 5/2 + 7/6
   ii. 1/2. + 5/2. + 3.
   iii. 1/2 + 5./2 + 2
   iv. (unsigned char) 500 + 200
   v. (unsigned char)(500 + 200)
   vi. (unsigned int) (unsingned char) 750

四、编程题

1. 编程实现：由键盘输入两个整数，然后输出最大者。
2. 假设 int a=5, b=0; 编程计算 a 和 b 的值是多少。
   (1) ! a && a+b && a++
   (2) !a||a++||b++
3. 有如下的定义 int a=4, b;指出下面表达式运算后 a 和 b 的结果?
   b+=b=++a;
4. 编程实现：测试你机器的 int、float、double、long、char 各类型变量存储的字节数。

第 3 章　流程控制语句

3.1　程序的基本控制结构

3.1.1　语句的分类

一个 C++源程序可以由若干个源程序文件组成，一个源程序文件可以由若干个函数和编译预处理命令组成，一个函数由函数说明部分和函数执行部分组成，函数执行部分由数据定义和若干执行语句组成。语句是组成程序的基本单元。C++中的语句分为以下几种。

1．说明语句

说明语句是对变量、符号常量、数据类型的定义性说明。

例如：`int a, b, c;　　//定义整型变量 a、b、c`

编译系统对说明信息，在程序执行期间并不执行任何操作。例如，定义变量语句“int a,b,c;”是告诉编译系统为变量 a、b、c 分配 12 字节的存储空间用于存放变量的值。程序执行时，该语句就不起任何作用了。

说明语句可出现在函数内、外，允许出现在程序的任何地方。

2．表达式语句

表达式语句是最简单的语句形式，一个表达式后面加上分号就构成了表达式语句，一般格式为：

```
表达式;
```

例如，赋值表达式可以构成赋值语句：

```
a=5;
```

3．空语句

只由一个分号构成的语句称为空语句。空语句不执行任何操作，但具有语法作用，例如 for 循环在有些情况下循环体是空语句，有些情况下循环条件判别是空语句，这些将在下节的循环语句中介绍。大多数情况下，从程序结构的紧凑性与合理性角度考虑，尽量不要随便使用空语句。

4．复合语句

由一对“{}”括起来的一组语句构成复合语句。复合语句描述一个块，在语法上起一个语句的作用。

对单个语句，必须以“;”结束，对复合语句，其中的每个语句仍以“;”结束，而整个复合语句的结束符为“}”。

5．流程控制语句

流程控制语句用来控制或改变程序的执行方向。

3.1.2　结构化程序控制结构

在程序的所有流程控制方式中，有三种是最基本的，即顺序控制、条件分支控制和循环控制；每种控制都有赖于一种特定的程序结构来实现，因此也就有三种基本的程序结构：顺序结构、条件分支结构和循环结构。

(1) 顺序结构：按语句的先后顺序执行。

(2) 条件分支结构：由特定的条件决定执行哪个语句的程序结构。可进一步分为单分支结构和多分支结构，在 C++中用 if 语句和 swich 语句实现。

(3) 循环结构：由特定的条件决定某个语句重复执行次数的控制方式。可进一步分为先判断后执行和先执行后判断。在 C++中用 while 语句、for 语句和 do…while 语句实现。

3.2　流程控制语句

3.2.1　if语句

if 语句称为分支语句，或条件语句，其功能是根据给定的条件，选择程序的执行方向。if 语句的基本格式为：

```
if（表达式） 语句1;
else  语句2;
```

其中的表达式称为条件表达式，可以是 C++中的任意合法表达式，如算术表达式、关系表达式、逻辑表达式或逗号表达式等。语句 1 和语句 2 也称为内嵌语句，在语法上各自表现为一个语句，可以是单一语句，也可以是复合语句，还可以是空语句。该语句的执行流程是，先计算表达式的值，若表达式的值为真(或非 0)，则执行语句 1，否则(表达式的值为假，或为 0)，执行语句 2。

分支语句在一次执行中只能执行语句 1 或语句 2 中的一个。如果语句 2 是空语句，else 也可以省略。这种情况下当条件表达式的值为假时，将不产生任何操作，直接执行分支语句之后的语句。例如，对于下列分支函数：

$$y=\begin{cases}0 & x<0\\ x^3+3x & x\geqslant 0\end{cases}$$

用 if 语句可以描述为：

```
if（x<0） y=0;
else  y=x*x*x+3*x;
```

也可以这样描述：

```
y=0;
if（x>=0） y=x*x*x+3*x;
```

这种描述的思想是，令 y 的值为 0，如果 x>=0，重新计算 y 的值，否则(即 x<0)y 的值不变。

【例 3.1】 输入一个年份，判断是否为闰年。

分析：假定年份为 year，闰年的条件是：year%4==0&&year%100!=0||year%400==0。

```
#include <iostream>
using namespace std;
void main(){
    int year;
    cout<<"输入年份:"<<endl;
    cin>>year;
    if (year%4==0&&year%100!=0||year%400==0)cout<<year<<"是闰年"<<endl;
    else  cout<< year<<"不是闰年"<<endl;
}
```

运行结果：

输入年份：

1900

1900 不是闰年

【例 3.2】 从键盘上输入 3 个整数，输出其中的最大数。

分析：输入三个整数，先求出两个数中较大者，再将该较大者与第三个数比较，求出最大数。

```
#include <iostream>
using namespace std;
void main(){
    int a, b, c, max;
    cout<<"输入三个整数:";
    cin>>a>>b>>c;
    cout<<"a="<<a<<'\t'<<"b="<<b<<'\t'<<"c="<<c<<endl;
    if(a>b) max=a;
    else max=b;
    cout<<"最大数为:";
    if(c>max) cout<<c<<endl;
    else cout<<max<<endl;
}
```

运行结果：

输入三个整正数：

2 9 6

a=2 b=9 c=6

最大数为：9

if 语句中，如果内嵌语句又是 if 语句，就构成了嵌套 if 语句。if 语句可实现二选一，而嵌套 if 语句则可以实现多选一的情况。嵌套有两种形式，一种是嵌套在 else 分支中，格式为：

```
if (表达式1)  语句1;
    else if (表达式2) 语句2;
else if …
    else 语句n;
```

第二种是嵌套在 if 分支中，格式为：

if（表达式 1）if（表达式 2）　语句 1；
else　语句 2；

【例 3.3】用嵌套 if 语句完成【例 3.2】的任务。

方法 1：采用第二种嵌套形式。

```
#include <iostream>
using namespace std;
void main( ){
   int a, b, c, max;
   cout<<"输入三个整数:";
   cin>>a>>b>>c;
   cout<<"a="<<a<<'\t'<<"b="<<b<<'\t'<<"c="<<c<<endl;
   if(a>b)  if(a>c)  max=a;          //a>b 且 a>c
          else  max=c;               //a>b 且 a<c
   else     if(b>c)  max=b;          //b>=a 且 b>c
          else  max=c;               //b>=a 且 b<c
   cout<<"最大数为:max="<<max<<endl;
}
```

运行结果：

输入三个整数：
3　7　12
a=3　b=7　c=12
最大数为：max=12

方法 2：采用第一种嵌套形式。

```
#include <iostream>
using namespace std;
void main( ){
   int a,b,c,max;
   cout<<"输入三个整数:";
   cin>>a>>b>>c;
   cout<<"a="<<a<<'\t'<<"b="<<b<<'\t'<<"c="<<c<<endl;
   if(a>b&&a>c)  max=a;
   else if(b>a&&b>c)  max=b;
      else  max=c;
   cout<<"最大数为:max="<<max<<endl;
}
```

运行结果：

输入三个整数：
8　1　5
a=8　b=1　c=5
最大数为：max=8

嵌套 if 语句同样可以默认任何一个 else，这时要特别注意 else 和 if 的配对关系。C++规定了 if 和 else 的“就近配对”原则，即相距最近且还没有配对的一对 if 和 else 首先配对。按

上述规定，第二种嵌套形式中的 else 应与第 2 个 if 配对。如果根据程序的逻辑需要改变配对关系，则要使用块的概念，即将属于同一层的语句放在一对“{}”中。如第二种嵌套形式中，要让 else 和第一个 if 配对，语句必须写成：

```
if（表达式 1）{
   if（表达式 2） 语句 1 ;
      }
   else 语句 2 ;
```

请看以下两个语句：

```
//语句 1:
if（n%3==0）
if（n%5==0） cout<<n<<"是 15 的倍数"<<endl;
else  cout<< n<<"是 3 的倍数但不是 5 的倍数"<<endl;
//语句 2:
if（n%3==0）{
if（n%5==0） cout<<n<<"是 15 的倍数"<<endl;
}
else  cout<< n <<"不是 3 的倍数"
```

两个语句的差别只在于一个“{}”，但表达的逻辑关系却完全不同。可以看出第二种嵌套形式较容易产生逻辑错误，而第一种形式配对关系则非常明确，因此从程序可读性角度出发，建议尽量使用第一种嵌套形式。

【例 3.4】 某商场优惠活动规定，某商品一次购买 5 件以上(包含 5 件)10 件以下(不包含 10 件)打 9 折，一次购买 10 件以上(包含 10 件)打 8 折。设计程序，根据单价和客户的购买量计算总价。

```
#include <iostream>
using namespace std;
void main(){
   float price,discount,amount;            //单价、折扣、总价
   int count;                              //购买件数
   cout<<"输入单价:"<<endl;
   cin>>price;
   cout<<"输入购买件数:"<<endl;
   cin>>count;
   if(count<5)  discount=1;
   else if(count<10)  discount=0.9;
       else  discount=0.8;
   amount=price*count*discount;
   cout<<"单价:"<< price<<endl;
   cout<<"购买件数:"<<count<<"\t\t"<<"折扣:"<<discount<<endl;
   cout<<"总价:"<<amount<<endl;
}
```

运行结果：

输入单价：

80

输入购买件数：
8
单价：80
购买件数：8　　折扣：0.9
总价：576

【例 3.5】 求一元二次方程 $ax^2 + bx + c = 0$ 的根。其中系数 a(a≠0)、b、c 的值由键盘输入。

分析：输入系数 a(a≠0)、b、c 后，令 delta = b^2–4ac，若 delta = 0，方程有两个相同实根；若 delta>0，方程有两个不同实根；若 delta<0，方程无实根。

```
#include <iostream>
#include <math.h>
using namespace std;
void main( ){
    float a,b,c;
    float delta,x1,x2;
    const float zero=0.0001;            //定义一个很小的常数
    cout<<"输入三个系数 a(a!=0), b, c:"<<endl;
    cin>>a>>b>>c;
    cout<<"a="<<a<<'\t'<<"b="<<b<<'\t'<<"c="<<c<<endl;
    delta=b*b-4*a*c;
    if(fabs(delta)<zero){               //绝对值很小的数即被认为是 0
        cout<<"方程有两个相同实根:";
        cout<<"x1=x2="<<-b/(2*a)<<endl;
    }
    else if(delta>0){
        delta=sqrt(delta);
        x1=(-b+delta)/(2*a);
        x2=(-b-delta)/(2*a);
        cout<<"方程有两个不同实根:";
        cout<<"x1="<<x1<<'\t'<<"x2="<<x2<<endl;
    }
    else                                //delta<0
        cout<<"方程无实根!"<<endl;
}
```

运行结果：

输入三个系数 a(a! = 0), b, c：
3　6　2
a=3　b=6　c=2
方程有两个不同实根：x1=–0.42265　x2=–1.57735

程序中有一个需要说明的问题，在判断 delta 是否为 0 时不是直接用表达式 delta==0，而是定义一个很小的常数 zero，用表达式 fabs(delta)<zero 进行判断。这是程序中经常遇到的关于实数判断的问题。由于实数在计算机中用浮点数表示，只能是近似值，即两个实数是不会

精确相等的。因此在判断两个实数是否相等时，比较规范的方法是用两数误差小于一个很小的数的方法。例如，两实数 x 和 y 如果满足表达式 fabs(x–y)<1e–4，则认为 x 等于 y。但在 C++中，由于运算中使用的是双精度数，精度足够高，因此大多数情况下实数也可以直接判断，即对于本例也可以用 delta==0 进行判断。

3.2.2 switch语句

用嵌套 if 语句可以实现多选一的情况。另外 C++中还提供了 switch 语句，称为开关语句，也可以用来实现多选一。它根据给定条件从多个分支语句序列中选择一个语句序列作为入口开始执行。格式为：

```
switch(表达式) {
      case 常量表达式1: <语句序列1;> <break;>
      case 常量表达式2: <语句序列2;> <break;>
      …
      case 常量表达式n: <语句序列n;> <break;>
            <default: 语句序列>
            }
```

其中表达式作为判断条件，称为条件表达式，取值为整型、字符型、布尔型或枚举型，关于枚举类型稍后介绍。各常量表达式是由常量构成的表达式，类型与条件表达式相同。各语句序列可以是一个语句，也可以是一组语句。

开关语句的执行过程是：先求条件表达式的值，并在常量表达式中找与之相等的分支作为执行入口，并从该分支的语句序列开始执行下去，直到遇到 break 语句或开关语句的花括号“}”为止。当条件表达式的值与所有常量表达式的值均不相等时，若有 default 分支，则执行其语句序列；否则跳出 switch 语句，执行后续语句。

关于 switch 语句，有几点需要注意：

(1) 各个 case(包括 default)分支出现的次序可以任意，通常将 default 放在最后。

(2) break 语句可选，如果没有 break 语句，每个 case 分支都只作为开关语句的执行入口，执行完该分支后，还将接着执行其后的所有分支。因此，为保证逻辑的正确实现，通常每个 case 分支都与 break 语句联用。

(3) 每个常量表达式的取值必须各不相同，否则将引起歧义。

(4) 允许多个常量表达式对应同一个语句序列。例如：

```
char score;
cin>>score;
switch (score) {
case 'A': case 'a':  cout<<"excellent"; break;
case 'B': case 'b':  cout<<"good"; break;
default: cout<<"fair";
}
```

(5) 从形式上看，switch 语句的可读性比嵌套 if 语句好，但不是所有多选一的问题都可由开关语句完成，这是因为开关语句中限定了条件表达式的取值类型。但在有些情况下，尽管条件表达式本身不符合数据类型的要求，但经过处理后便可用开关语句实现。

【例 3.6】 运输公司对所运货物实行分段计费。设运输里程为 s，则运费打折情况如下：

s<250	不打折扣
250<=s<500	2%折扣
500<=s<1000	5%折扣
1000<=s<2000	8%折扣
2000<=s<3000	10%折扣
3000<=s	15%折扣

设每公里每吨的基本运费为 p，货物重量为 w，折扣为 d，则总运费 f 为：

f=p*w*s*(1-d)

设计程序，当输入 p、w 和 s 后，计算运费 f。

分析：如果用 switch 语句，必须使表达式符合语法要求。分析发现，里程 s 的分段点均是 250 的倍数，因此将里程 s 除以 250，取整数商，便得到若干整数值。

```
#include <iostream>
#include <iomanip >
using namespace std;
void main( ){
   int c,s;
   float p,w,d,f;
   cout<<"输入运输单价 p，重量 w 和里程 s:"<<endl;
   cin>>p>>w>>s;
   c=s/250;
   switch(c){
   case 0:                              d=0;   break;
   case 1:                              d=0.02; break;
   case 2: case 3:                      d=0.05; break;
   case 4: case 5: case 6:  case 7:     d=0.08; break;
   case 8: case 9: case 10: case 11:    d=0.1;  break;
   default:                             d=0.15;
}
    f=p*w*s*(1-d);
   cout<<"运输单价:"<<p<<'\t'<<"重量:"<<w<<'\t'<<"里程:"<<s<<endl;
   cout<<"折扣:"<<d<<endl;
   cout<<"运费:"<<f<<endl;
}
```

运行结果：

输入运输单价 p、重量 w 和里程 s：

50 150 460

运输单价：50　　重量：150　　里程：460

折扣：0.02

运费：3.381e+006

【例 3.7】 设计一个计算器程序，实现加、减、乘、除运算。

分析：读入两个操作数和运算符，根据运算符完成相应运算。

```
#include <iostream>
using namespace std;
```

```
void main(){
    float num1,num2;
    char op;
    cout<<"输入操作数 1，运算符，操作数 2:"<<endl;
    cin>>num1>>op>>num2;
    switch(op){
    case '+':  cout<<num1<<op<<num2<<"="<<num1+num2<<endl; break;
        case '-':  cout<<num1<<op<<num2<<"="<<num1-num2<<endl; break;
        case '*':  cout<<num1<<op<<num2<<"="<<num1*num2<<endl; break;
        case '/':  cout<<num1<<op<<num2<<"="<<num1/num2<<endl; break;
        default :  cout<<op<<"是无效运算符!";
}
}
```

运行结果：

输入操作数 1，运算符，操作数 2：

3 * 7

3*7=21

3.3 循环控制语句

3.3.1 for循环

for 语句也称 for 循环，语句格式为:

```
for (表达式 1; 表达式 2; 表达式 3)  循环体语句
```

该语句的执行过程是，先求表达式 1 的值，再求表达式 2 的值，如果表达式 2 的值为真(或非 0)，则执行循环体语句，并求表达式 3 的值，然后再计算表达式 2 的值，并重复以上过程，直到表达式 2 的值为假(或为 0)，结束该循环语句。图3-1是 for 语句的执行流程。

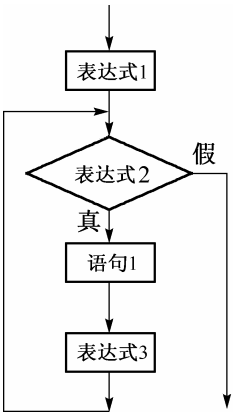


图 3-1 for 语句的执行流程

关于 for 语句，有以下几点说明:

(1) 从执行流程看，for 语句属于先判断型，因此与 while 语句是完全等同的。

(2) for 语句中的三个表达式都是包含逗号表达式在内的任意表达式。从逻辑关系看，循环初始条件可在表达式 1 中给出，循环条件的判断可包含在表达式 2 中，而循环条件变量的修改可包含在表达式 3 中，也可以放在循环体中。如例 3.8 中的循环部分用 for 语句可描述为：

```
for (i=1, s=0; i<=100; i++)  sum+=i;
```

(3) for 语句中的三个表达式可部分或全部省略，但两个分号不能省略。

从执行流程看，如果表达式 1 放在 for 语句之前，或表达式 3 放在循环体中，那么相应地在 for 语句中就可省略表达式 1 或表达式 3。如上述语句还可写为：

```
i=1; sum=0;
for ( ;i<=100; ) {
```

```
sum+=i;
i++;
}
```

实际上，表达式 2 也可省略。如果表达式 2 省略，系统约定其值为 1，这种情况表达式 2 的值恒为真，即：

```
for (; ;) {…}
```

等同于

```
for (; 1;) {…}
```

这种形式的循环如不加控制将导致死循环。为避免死循环，循环体内必须用 break 语句来终止循环，具体方法稍后介绍。

【例 3.8】 古罗马数学家伦纳德・斐波那契提出一个有趣的问题。假定每对兔子每月生出一对小兔子，新生的一对小兔子三个月后又可以生小兔子，假定所有兔子都不会死，一年后会有多少对兔子。具体说，第一个月只有一对兔子，第二个月由于新生小兔子不能生育，仍然只有一对兔子，第三个月小兔子开始生育，因此当月有两对小兔子，此后每个月的兔子数都是上个月和当月新生兔子数之和。由此可抽象出一个数列：0, 1, 1, 2, 3, 5, 8, …。这个数列称为 Fibonacii 数列，可用函数描述如下：

$$\text{fib}(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ \text{fib}(n-2)+\text{fib}(n-1) & n>1\end{cases}$$

设计程序输出 Fibonacii 数列的前 20 项，要求每行输出 5 个数据。

```
#include<iostream>
#include<iomanip>
using namespace std;
const int m=20;
void main( ){
    int fib0=0,fib1=1,fib2;
    cout<<setw(15)<<fib0<<setw(15)<<fib1;
    for(int n=3;n<=m;n++){
        fib2=fib0+fib1;
        cout<<setw(15)<<fib2;
        if(n%5==0)  cout<<endl;          //控制每行 5 个数据
        fib0=fib1;  fib1=fib2;
    }
}
```

运行结果：

```
0              1              1              2              3
5              8             13             21             34
55            89            144            233            377
610           987           1597           2584           4181
```

【例 3.9】 输入一个不超过 9 位的整数，将其反向后输出。例如，输入 247，变成 742 输出。分析：将整数的各个数位逐个分开，用一个数组保存各个位的值，然后反向组成新的整

数。将整数各位数字分开的方法是，通过求余得到个位数，然后将整数缩小为 1/10，再求余，并重复上述过程，分别得到十位、百位……，直到整数的值变成 0 为止。

```
#include <iostream>
using namespace std;
void main(){
    int num,subscript;
    int digit[9];
    cout<<"输入一个整数:"<<endl;
    cin>>num;
    cout<<"原来整数为:"<<num<<endl;
    subscript=0;                          //数组下标初值
    do{
        digit[subscript]=num%10;
        num=num/10;
        subscript++;                      //修改下标
} while (num>0);
    for(int i=0;i<subscript;i++)          //整数的反向组合
num=num*10+digit[i];
    cout<<"反向后整数为:"<<num<<endl;
}
```

运行结果：

输入一个整数：
375
原来整数为：375
反向后整数为：573

3.3.2 do while循环

do-while 语句格式为：

do 循环体语句 while(表达式)

该语句的执行过程是，执行一次循环体语句，然后计算表达式的值，若表达式的值为真(或非 0)，则重复上述过程，直到表达式的值为假(或为 0)时结束循环。图3-2给出该语句的执行流程图。

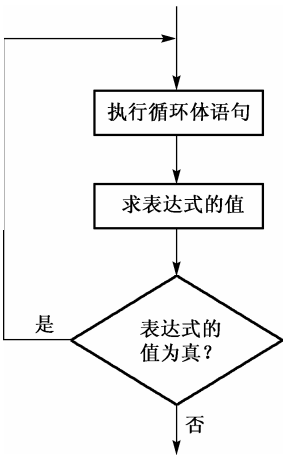


图 3-2 do-while 语句的执行流程图

do-while 语句在绝大多数情况下都能代替 while 语句，两个语句之间的区别是，do-while 语句无论条件表达式的值是真是假，循环体都将至少执行一次；而 while 语句如果条件表达式的初值为假，则循环体一次也不会执行。

【例 3.10】 用迭代法求 $x=\sqrt{a}$ 的近似值。求平方根的迭代公式为：

$$x_{n+1}=(x_n+a/x_n)/2$$

要求前后两个迭代根之差小于 10^{-5}。

分析：这是递推算法的一个应用。从键盘读入一个正数赋给 a，人为估计一个值作为迭代

初值 x0，假定取 a/2，根据迭代公式求出 x1，若|x1–x0|<10^{-5}，则 x1 就是所求的平方根近似值；否则，将 x1 赋给 x0，再用公式迭代出新的 x1。重复以上过程直到|x1–x0|<10^{-5}为止。

```
#include<iostream>
#include<math>
using namespace std;
void main(){
    float x0,x1,a;
    cout<<"输入一个正数:"<<endl;
    cin>>a;
    if(a<0)  cout<<a<<"不能开平方!"<<endl;
    else {                                          //有实数解的情况
        x1=a/2;                                     //x1 用做保存结果
        do {
            x0=x1;
            x1=(x0+a/x0)/2;
        } while (fabs(x1-x0) >=1e-5);
        cout<< a<<"的平方根为:"<<x1<<endl;
}
}
```

运行结果：

输入一个正数：

5

5 的平方根为：2.23607

【例 3.11】 输入一段文本，统计文本的行数、单词数及字符数。假定单词之间以空格、跳格或换行符间隔，假定文本没有空行。

分析：通过本例介绍输入结束符 EOF。执行 cin.get()将返回字符的 ASCII 码，而当读入的字符为键盘上的“Ctrl+z”时，将返回一个整数–1，该值被定义为 EOF，因此可用这个符号作为文本输入的结束标志。该程序设计思想为，逐个读入文本中的字符，直到读到“Ctrl+z”为止。其中行结束标志为字符'\n'。先令行数 nline、单词数 nword、字符数 nch 的初值均为 0。在读入过程中，每读到一个非间隔符，nch++，每读到一个'\n'，nline++；另设一个变量 isword，作为是否读到单词的标志。读到字符时 isword=1，读到间隔符时 isword=0。如果读到一个间隔符而此时 isword 值为 1，表明一个单词结束，则 nword++。

```
#include<iostream>
using namespace std;
void main(){
    char ch;
    int nline=0,nword=0,nch=0;
    int isword=0;
    cout<<"输入一段文本(无空行):"<<endl;
    do{
        ch=cin.get();
        if(ch=='\n')  nline++;                          //遇换行符，行数+1
        if(ch!=' '&& ch!='\t'&&ch!='\n'&&ch!=EOF){  //读到非间隔符
```

```
            if(!isword)  nword++;                    //在单词的起始处给单词数+1
            nch++;                                   //字符数加+1
            isword=1;
        }
        else  isword=0;                              //读到间隔符
}while(ch!=EOF);                                     //读到文本结束符为止
cout<<"行数:"<<nline<<endl;
cout<<"单词数:"<<nword<<endl;
cout<<"字符数:"<<nch<<endl;
}
```

运行结果:

输入一段文本(无空行):

```
hello
start exercise
good work
<ctrl+z>
```

行数：3
单词数：5
字符数：22

3.3.3 while循环

while 语句格式为:

```
while (表达式) 循环体语句;
```

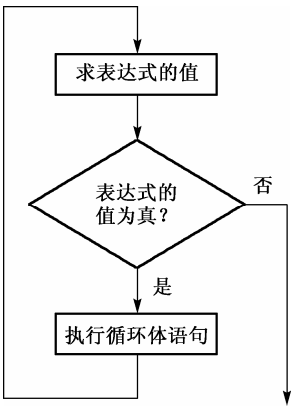


图 3-3 while 语句的执行流程图

其中表达式是 C++中任一合法表达式，包括逗号表达式；循环体语句可以是单一语句，也可以是复合语句。while 语句的执行过程是，先计算表达式的值，如果值为真(或非 0)，则执行循环体，然后再计算表达式的值，并重复以上过程，直到表达式的值为假(或为 0)，便不再执行循环体，循环语句结束。图3-3给出该语句的执行流程图。

【例 3.12】 计算 1+2+3+…+100 的值。

分析：计算累加和实际上是重复一个循环，在循环中将下一个数与累加和相加。

```
#include <iostream>
const int n=100;        //采用常变量有利于修改程序
void main(){
    int i=1,sum=0;      //循环初始条件
    while(i<=n){
        sum+=i;
        i++;            //修改循环条件
}
    cout<<"sum="<<sum<<endl;
}
```

运行结果:

sum=5050

在有循环语句的程序中，通常在循环开始前对循环条件进行初始化，如上例中的 i=1；在循环语句中要包含修改循环条件的语句，如例 3.12 中的 i++，否则循环将不能终止而陷入死循环。

C++表达方式灵活，上例中的循环语句还可以写成：

```
while（i<=n） sum+=i++;
```

或者

```
while（sum+=i++, i<n）;          //循环体为空语句
```

需要说明的是，虽然 C++可以让代码最大限度优化，但往往造成可读性降低，因此程序设计者只需理解这种表达方法的意义，而设计时主要追求的目标应是可读性。

3.4　跳转语句

通常程序语句都是顺序执行的，包括循环语句和分支语句也是按照语句的语法要求顺序执行相应的操作。C++还提供若干跳转语句，可以改变程序原来的执行顺序。

3.4.1　break语句

break 语句只能用在 switch 语句和循环语句中，从 break 语句处跳出 switch 语句或循环语句，转去执行 switch 语句或循环语句之后的语句。break 语句在 switch 语句中的用法前面已经介绍过，在循环语句中用来提前终止循环，例如：

```
for（i=10; i<20; i++）{
if（i % 3==0）break;
cout<<i<<'\t';
}
```

该程序段执行后将输出：

10　11

前面介绍过，for 语句的三个表达式均可以是空语句，在这种情况下，for 循环必然是如下形式：

```
for(; ;) {
…
if(表达式) break;
…
}
```

否则将导致死循环。可见 break 语句的参与使循环语句的使用更加灵活。

需要注意的是，在嵌套循环中，break 语句终止的是其所在的循环语句，而并非终止所有的循环。例如：

```
for（; ;）{
for（; ;）{
…
… break;
```

```
…
}
语句 1 ;
…
}
```

当程序执行到 break 语句时，终止的是内层循环，接着执行语句 1 。

【例 3.13】 给定正整数 m，判定其是否为素数。

分析：如果 m>2，m 是素数的条件是不能被 2、3、…、($\sqrt{m}$ 取整)整除。因此可以用 2、3、…、($\sqrt{m}$ 取整)逐个去除 m，如果被其中某个数整除了，则 m 不是素数，否则是素数。

```
#include <iostream>
#include <math>
using namespace std;
void main( ){
    int m,i,k;
    cout<<"输入整数 m:"<<endl;
    cin>>m;
    if(m==2)  cout<<m<<"是素数"<<endl;
    else{
        k=sqrt(m);
        for(i=2;i<=k;i++) if (m%i==0) break;    //只要有一个整除，就可停止
        if(i>k) cout<< m<<"是素数"<<endl;       //循环提前终止表示是非素数
        else  cout<< m<<"不是素数"<<endl;
    }
}
```

执行结果为：

```
输入整数：
35
35 不是素数
```

3.4.2 continue语句

continue 语句只能用在循环语句中，用来终止本次循环。当程序执行到 continue 语句时，将跳过其后尚未执行的循环体语句，开始下一次循环。下一次循环是否执行仍然取决于循环条件的判断。例如：

```
for (i=10; i<20; i++) {
if (i % 3==0) continue;
cout<<i<<'\t';
}
```

该程序段执行后将输出：

```
10   11   13   14   16   17   19
```

continue 语句与 break 语句的区别在于，continue 语句结束的只是本次循环，而 break 结束的是整个循环。

3.4.3　goto语句

goto 语句又称转向语句，其功能是令程序跳转到程序指定的某标号语句处。由于标号语句的设定较为灵活，因此，goto 语句的跳转控制转语句有较大的随意性。其格式为：

```
goto〈标号〉;
```

goto：关键字 goto 指明该语句为 goto 语句，其后必须有一标号。

标号：标号是一个标识符，其定义出现在标号语句：

```
〈标号〉:〈语句〉
```

例如：

```
…
L1:〈语句 S1〉
…
goto L2;
…
L2:〈语句 S2〉
…
goto L1;
…
```

语句“goto L2;”把控制流程跳转到“L2:〈语句 S2〉”；而语句“goto L1”，又跳转到“L1:〈语句 S1〉”。标号语句可以出现在转向它的 goto 语句之后，也可以出现在其之前。goto 语句和 break、continue、return 语句都是无条件转移语句，无须任何判断，程序执行到此时，必须跳转到一个确定的位置。不过，goto 语句与前三者不同的是：goto 语句转向的目标位置由程序员任意确定，而 break、continue 和 return 语句的转向目标位置唯一地由其本身的位置所决定。然而，goto 语句的转向目标——标号语句的位置虽然可由程序员确定，但不是完全任意的，它应满足下面的限制：

(1) 函数定义(函数体)内的 goto 语句不可转向到函数之外。换句话说，函数的出口点只能是 return 语句或函数体结束。

(2) 块语句(包括函数体和循环体)外的 goto 语句不可转向到该程序块之内，因为这往往会产生计算机难于处理的局面。即使有这样的限制，goto 语句仍然是十分自由的，goto 语句的使用容易造成：

① 使程序的静态结构与程序的动态结构差别增大，使程序段之间形成“交叉”的关系，不利于程序的维护和调试。

② 一个好的程序，它的各个程序段最好是单入口单出口的，没有死循环和死区(不可到达的程序段)。但 goto 语句的使用容易破坏这种状态。

③ 历史上关于 goto 语句的讨论是程序设计和软件开发史上的重大事件之一，最近一次的讨论，是由“goto 语句是有害的”这样一篇著名论文引起，最终以“限制使用 goto 语句”作为结论，从而对软件开发的发展产生了重要影响。

因此，本书向读者建议，不用或少用 goto 语句。但在某些特定场合下，goto 语句可能会

显出其价值，如在多层循环嵌套中，要从深层地方跳出所有循环，如果用 break 语句，不仅要使用多次，而且可读性较差，这时 goto 语句可以发挥作用。

3.4.4 return语句

return 语句又称返回语句。只用于函数定义，其功能为:

(1) 把程序运行的流程跳转到该函数的调用点或函数调用的出口点。

(2) 计算返回表达式 E，并把其值作为该函数的返回值。其格式为：

```
return;
```

或

```
return〈表达式 E〉;
```

return：关键字 return 仅用于返回语句。

表达式 E：当函数的返回类型为 void 型时，返回语句应取第一种方式，否则取第二种方式。表达式 E 的类型应与函数的(返回)类型相一致。对于非 void 类型的函数来说，其函数体中的 return 语句是必不可少的。

本 章 小 结

本章介绍了 C++的流程控制语句，包括分支语句：if、if-else 语句，多分支 switch 语句；循环语句：for、while、do-while 语句；跳转语句:break、continue、goto 及 return 语句。

if 语句与 switch 语句可以互换使用。对于多分支的情形，常用 switch 语句，如用 if-else 语句，会使程序显得复杂，可读性较差。

与 while 和 do-while 语句相比，for 语句的使用更为灵活，既可用于循环次数已确定的情况，也可用于循环次数不确定而只给出循环结束条件的情况。在表达方面，三个表达式及循环体都可以灵活使用，甚至省略，因此 for 语句是三个循环语句中用得最多的一个。另外要注意 while 循环语句与 do-while 的差别：do-while 循环语句至少要执行一次循环体。编程时，常常是循环可能一次也不执行，此时就不能用 do-while 循环语句，而要用 while 循环语句或 for 循环语句。

习 题 3

一、选择题

1. 如果变量 x, y 已经正确定义，下列语句哪一项不能正确将 x、y 的值进行交换:

A. x=x+y, y=x-y, x=x-y;　　B. t=x, x=y; y=t;

C. t=y, y=x, x=t;　　D. x=t, t=y, y=x

2. 如要求在 if 后一对括号中的表达式，表示 a 不等于 0 的时候的值为“真”，则能正确表示这一关系的表达式为：

A. a<>0　　B. !a　　C. a=0　　D. a

3. 下面的循环的循环次数是：for(int i=0, j=10; i=j=10; i++, j--)

A. 无限次　　B. 语法错误，不能执行

C. 10　　D. 1

4. 以下哪个不是循环语句：

A. while 语句　　B. do-while 语句

C. for 语句　　D. if-else 语句

5. 下列 do-while 循环的循环次数是:

已知：`int i=5;`

```
do{cout<<i--<<endl;
i--;
}while(i!=0)
```

A. 0　　B. 2　　C. 5　　D. 无限次

6. 下列 do-while 循环的循环次数是：

for(int i=0,x=0;!x&&i<=5;i++)

A. 5　　B. 6　　C. 1　　D. 无限次

二、填空题

1. do-while 语句与 while 语句的主要区别是_____。

2. x、y、z 为 int 类型的时候，下列语句执行之后，x 的值为____，y 的值为____，z 的值为 _____。

```
x=10;y=20;z=30;
if(x>y) x=y;y=z;z=x;
```

3. _____是构造程序最基本的单位，程序运行的过程就是_____的过程。

4. break 语句实现的功能是_____。

5. continue 语句实现的功能是_____。

三、简答题

1. 用两次 for 循环和求余运算符测试和打印输出(2～100)中为素数的正整数。

2. 写一段程序，使用 while 循环从标准输入设备中读取字符到字符串中，自己确定输入什么字符时退出循环，对每个读取的字符先用 if 语句判别是否为数字，然后用 switch 语句判别是否为字母，并打印输出相应字母为首字母的英文单词到屏幕。退出循环后在屏幕上显示输入的字符串。

3. 读下段程序，分析其功能是什么?

```
#include <iostream>
using namespace std;
void main( )
{
  cout<<"please input the b key to hear a bell."<<endl;
  char ch;
  cin>>ch;
  if (ch=='b')
    cout <<'\a';
```

```
    else
      if (ch=='\n'
        cout <<"what a boring select on..."<<endl;
      else
        cout <<"bye! \n";
  }
```

四、编程题

1. 编写程序实现以下功能：从键盘读入 3 个整数，比较其大小，输出其中的最大数。
2. 编程实现：输入一行字符，求其中字母、数字和其他符号的个数。
3. 求满足下式的最小 n 值，其中 limit 由键盘输入。

   1+1/2+1/3+…+1/n>limit
4. 编写程序：输出菲波那切数列的前 20 项。即前两项为 1，以后每项为前两项之和。
5. 编写程序：根据学生成绩输出优、良、中、及格和不及格。
6. 编程实现：读入一行字母，求其中元音字母出现的次数。

第 4 章　数组和字符串

数组用于处理一组具有相同类型的数据，本章将介绍数组和字符串的概念、定义、存储与应用。通过示例学习利用数组和字符串对批量数据的表示、存储、计算、排序和查找等运算。

4.1　数组的概念

在前面章节中介绍了单个数据的使用，不同的数据类型需要定义相应的变量来保存。当存储含有多种类型的记录数据时，需要定义该类型的结构变量来保存。

在程序设计中，若需要存储同一数据类型且各数据间具有相关性的一组数据时，就要求能够同时存储多个值的变量，这种变量在程序设计中称为数组，同一个数组中的每个值通过下标来区分(标识)。如一天 24 小时，要求表示整点时间的温度，可以用 t1，t2，…，t24 表示。这样表示数据不仅处理起来方便，而且能更清楚地表示各数据之间的关系，这批数据就称为数组，用来表示这批数据的变量称为下标变量。

所以说数组是具有相同类型的数据按一定顺序组成的变量序列。数组中的每个数据都可以通过数组名及唯一的索引号(下标)来确定。

4.2　数组的定义和数组元素表示方法

数组必须先声明后使用。声明数组后，就可以对数组进行访问了。访问数组实质是对数组中某个元素或全部元素进行访问，即对数组中的元素进行读写操作。

在实际应用中，数据之间可能存在着一维关系，也可能存在着二维关系。对一维数组中的每个数据来说，它除第一个数据外，每个数据只有一个直接前驱；除最后一个数据外，每个数据只有一个直接后继。如数列(10，30，55，77，99)，则每个整数的后一个整数就是它的直接后继，每一个整数的前一个整数就是它的直接前驱。对二维数组中的每个元素来说，它除第一行和第一列上的所有数据外，每个数据在行和列的方向上各有一个直接前驱；除最后一行和最后一列上的所有数据外，每个数据在行和列的方向上各有一个直接后继。这样的数据就用二维数组来表示，例如：

3　4　7
1　5　2
8　3　9

则每个元素均有行、列之分。在 C++程序设计中，要求数组元素的下标必须放在一对方括号中，并紧跟在数组名之后。如前面表示温度的例子，必须写成 t[1]，t[2]，…，t[24]。C++语言规定：一维数组中元素的下标从常数 0 开始，以后依次增 1。如对含有 n 个元素的数组 a，则下标编号默认为 0，1，2，…，$n-1$，用 a[0]，a[1]，…，a[n–1]来存储数组中的每个数据。

对一个 $m \times n$ 阶的数组：$\begin{bmatrix} a_{11} & a_{12} & \text{L} & a_{1n} \\ a_{21} & a_{22} & \text{L} & a_{2n} \\ \text{M} & \text{M} & \text{O} & \text{M} \\ a_{m1} & a_{m2} & \text{L} & a_{mn} \end{bmatrix}$

则需要用一个二维数组来表示和存储。如用 b 表示，则 b 中应包含 $m \times n$ 个元素，第 1 维下标依次为 0，1，2，…，$m-1$，第 2 维下标依次为 0，1，2，…，$n-1$，数组中的第 1 个元素 a_{11} 被存储到 b[0][0]元素中，最后一个元素 a_{mn} 被存储到 b[m–1][n–1]元素中，依次类推。

4.2.1 数组的定义

1．一维数组的定义格式

```
<类型关键字> <数组名> [<常量表达式>] [={<初值表>}];
```

2．二维数组的定义格式

```
<类型关键字><数组名> [<常量表达式 1>] [<常最表达式 2>] [={[<初值表 1>},
{<初值表 2>}，…}];
```

其中：

<类型关键字>为已存在的一种数据类型；

<数组名>是用户定义的一个标识符；

[<常量表达式>]为一个整数值，用以说明数组的长度，即数组中所含元素的个数；

<初值表>是用逗号隔开的一组数据，每个数据的值将被赋给数组中相应的元素。<初值表>选项可以省略；<常量表达式>可以为空，此时所定义的数组的长度是<初值表>中所含表达式的个数。

二维数组定义中的<常量表达式 1>和<常量表达式 2>分别指定数组中第 1 维下标和第 2 维下标的个数。分别表示行与列的个数，假定用 m、n 表示行与列的个数，则该数组下标的取值范围是 0～m–1 和 0～n–1。

数组被定义后，系统将在内存中为它分配一块含有 n 个存储单元的存储空间，根据数据类型的不同，所占用的存储空间也不同。如定义 int a[10]，则该数组将占用 10×4=40 字节的存储空间。如定义 double b[3][4]，则该数组将占用 12×8 = 96 字节的存储空间。

3．三维数组的定义和使用

在 VC++ 2005 中，也可以定义和使用三维及更高维的数组。例如，下面的语句定义了一个三维数组：

```
int s[x][y][z];        //假定 x、y、z 均为已定义的整型常量
```

该数组的数组名为 s，第 1 维下标的取值范围为 0～x–1，第 2 维下标的取值范围为 0～y–1，第 3 维下标的取值范围为 0～z–1，该数组共包含 x×y×z 个 int 型的元素，共占用 x×y×z×4 字节的存储空间。若用一个三维数组来表示一本书中某页一个文字，则第 1 维表示页号，第 2 维表示某页中的行号，第 3 维表示某行中的列号。

4.2.2　格式举例

```
int a[20];                          //定义一个含 20 个整型元素的数组
double b[m];                        //定义一个双精度型数组，m 为常量
int c[5]={1,2,3,4,5};               //定义一个整型数组，其数组元素为 1、2、3、4、5
char d[]={'a','b','c','d'};         //定义一个字符数组，隐含数组元素个数为 4
int e[8]={1,4,7};                   //定义一个整型数组，前三个元素为1, 4, 7，其余为0
char f[10]={'C','H','I','N','A'};//定义一个字符数组，前 5 个为 C、H、I、N、A，
                                      第 6 个元素为'\0'
bool g[2*n];                        //定义一个布尔型数组，其中 n 为常量
float a[5],b[10];                   //定义两个单精度数组
int a[3][3];                        //定义一个整型的二维数组，共有 9 个数组元素
double b[m][n];                     //定义一个双精度型二维数组，m、n 为常量
int p[];                            //错误的数组定义，因为无法确定数组的大小
int a[ ][5];                        //错误的二维数组定义，因为第一维下标的取值无
                                      法确定
int c[2][4]={{1,3,5,7},{2,4,6,8}}; //定义一个二维整型数组，其二维数组形式为：
```

$$\begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{bmatrix}$$

4.3　数组元素的输入与输出

(1) 简单的输入、输出

数组元素如果是已知的，可以在定义时直接赋值，例如：

```
int a[5]={1,2,3,4,5};
```

数组中各元素定义完成后，就可用 cout 语句直接输出，例如：

```
for(i=0;i<5;i++) cout<<a[i] <<' ';      //输出结果为 1、2、3、4、5
```

也可以输出数组中的一个或几个元素，例如：

```
cout<<a[2]<<' '<<a[a[2]];               //输出结果为 3 和 4
```

(2) 利用循环语句输入、输出

如果数组中的元素值是未知的，可以先定义数组，然后利用 cin 和 cout 语句进行输入和输出，例如：

```
int a[6];
for(int i=0;i<6;i++)    cin>>a[i];
for(int i=5;i>=0;i--)   cout<<a[i];
```

如果是二维数组，则应该用双循环进行输入和输出，如给二维数组元素赋值：

```
int x[2][3];
for(int i=0;i<2;i++)
```

```
    for(int j=0;j<3;j++)
     cin>>x[i][j];
```

输出二维数组元素的值，例如：

```
int x[2][3]={1,3,4,7,9,8};
for(int i=0;i<2;i++)
       {for(int j=0;j<3;j++)
         cout<<x[i][j]<<' ';
         cout<<endl;
}
```

注意：在输出二维数组元素时，一定要在两个循环语句之间加大括号。

(3) 程序举例

【例 4.1】 从键盘上输入一组数据，并按相反的顺序打印出来。

```
#include <iostream>
using namespace std;
void main( )
{ int i,a[10];
  for(i=0;i<10;i++)
      cin>>a[i];
  for(i=9;i>=0;i--)
      cout<<a[i]<<' ';
  cout<<endl;
}
```

该程序首先定义了一个整型数组 a，利用循环语句给数组中各元素赋值，然后再用一个循环按序号从大到小的顺序输出。

【例 4.2】 利用随机函数求一组数中的最大值及其位置。

```
#include <iostream>
#include <ctime>                //调用系统时间
#include<cstdlib>               //调用随机数的函数
using namespace std;
void main( )
{ int i,max,k,a[10];
     srand((unsigned)time(0));  //使每次产生的随机数不同
  for(i=0;i<10;i++)
      a[i]=rand( )%1000;        //保证产生的随机数在 1000 以内
max=a[0];
k=0;
  for(i=0;i<10;i++)
      if (a[i]>max){max=a[i],k=i;}
for(i=0;i<10;i++)
      cout<<a[i]<<' ';          //输出数组中个元素的值
cout<<endl;
cout<<"The location of the maximum is:"<<k<<"   Values are:"<<max;
cout<<endl;
}
```

在这个程序中首先利用随机函数给 a 数组赋值，然后初始化 max 和 k 变量，利用循环语句将数组中的每个元素与 max 比较，并利用变量 k 记录最大值的位置。

该程序的运行结果为：

```
352   834   583   779   956   834   103   969   245   109
The location of the maximum is: 7     Values are:969
```

【例 4.3】 输出数列中的前 10 项：1，1，2，3，5，8，13，…

```
#include <iostream>
using namespace std;
void main( )
{ int i,a[10];
a[0]=1;a[1]=1;
for(i=2;i<10;i++)
      a[i]=a[i-1]+a[i-2];
  for(i=0;i<10;i++)
      cout<<a[i]<<' ';
cout<<endl;
}
```

因为数列中前 2 项的值是已知的，所以第 5 行是为数列的前 2 项赋初始值，从第 3 项开始(下标从 2 开始)每一项是前 2 项的和，第 6 行利用循环语句将第 3～10 项的值求出来，最后再用循环语句将前 10 个数都打印出来。

【例 4.4】 求 n×n 矩阵中主对角线元素的乘积。

```
#include <iostream>
using namespace std;
const int n=4;
void main( )
{ int i,j,a[n][n];
int p=1;                  //初始化主对角线乘积变量
for(i=0;i<4;i++)
for(j=0;j<4;j++)
     cin>>a[i][j];
  for(i=0;i<4;i++)        //求主对角线元素的乘积
p*=a[i][i];
     cout<<"主对角线元素的乘积是"<<p;
cout<<endl;
}
```

该题目中并未说明 n 的大小，所以先定义 n 是一个整型常量；然后确定 a[n][n]的大小；第 7、8 行双循环语句给数组 a 赋值；由于对角线上的元素特点是行数和列数相等，所以第 10 行用单循环语句求出主对角线元素的乘积。

4.4　数组的应用

前面的题例中介绍了数组的一些简单应用，在现实生活中程序员还经常利用数组中的数据进行统计、排序、查找、计算等。下面通过题例来说明这些运算。

4.4.1 统计

【例 4.5】 在一次选举学生会主席的大会上，有 5 名候选人，分别用 1、2、3、4、5 代表每位候选人的号码，统计出每人的得票数。用–1 作为终止输入的标志。

程序如下：

```
#include <iostream>
using namespace std;

void main( )
{  int i,a[6]={0};
cout<<"请输入每张票上所投候选人的代号";
cin>>i;
while(i!=-1){
   if(i>=1 && i<=5)a[i]++;
   cin>>i; }
cout<<endl;
for(i=1;i<=5;i++)
cout<<i<<":"<<a[i]<<endl;
}
```

在程序中用数组 a 作为计数器统计每位候选人的得票数；由于题目中未说明参选人数，因此使用 while 循环；用 if 语句保证只统计 1～5 的有效数字。

假设从键盘输入的数字为：2 3 4 1 5 3 2 2 1 1 1 1 1 –1

结果为：

1:6
2:4
3:2
4:1
5:1

【例 4.6】 某社区对居民用电情况进行统计，每隔 50 度为一个统计区域，当大于等于 500 度时不再统计，编一程序，分别统计各用电区间内的居民数。

```
include <iostream>
using namespace std;
const int n=100;
void main( )
{   int a[11]={0};
   int i,x;
   for(i=1;i<=n;i++)
   {   cin>>x;
      if(x<500)a[x/50]++;
      else a[10]++;
   }
   for(i=0;i<=10;i++)
   cout<<"a["<<i<<"]="<<a[i]<<endl;
}
```

在题目中提示每隔 50 度为一个统计区域，从 0 度到 500 度可以划分 11 个区域，因此用 a[11]={0}作为统计居民用电情况的计数器，并在定义时赋初值零；程序中假定用户数为 100，第 9 行的 if(x<500)a[x/50]++语句根据题目要求直接将符合条件的居民用电情况放到计数器中累加。

4.4.2　排序

1．选择法排序

选择法排序的基本思想是:对待排序的序列进行若干遍处理，通过 $n-i$ 次关键字的比较，从 $n-i+1$ 个数据中选出最小的数据和第 $i(1 \leqslant i \leqslant n)$个数据进行交换，这样一遍处理就能确定一个数的位置，对 n 个数如果经过 $n-1$ 次处理，则这 n 个数就排序成功。

【例 4.7】 已知有 10 个整数，采用选择法将这 10 个数按从小到大的顺序打印输出。

```
#include <iostream>
using namespace std;
void main( )
{ int a[10]={33,61,43,74,86,92,11,35,64,25};
  int i,j,k;
  for(i=1;i<10;i++)          //外循环控制次数
  {  k=i-1;                  //外循环每一遍时最小值的位置
     for(j=i-1;j<10;j++)     //外循环每一遍时需要比较数据的次数
       if(a[j]<a[k])k=j;     //将最小值的位置号记录下来
       int x=a[i-1];a[i-1]=a[k];a[k]=x;  //将当前数与最小值互换
  }
for(int i=0;i<10;i++)cout<<a[i]<<"   ";  //输出排序后的数据
cout<<endl;
}
```

该程序的运行结果为：

11 25 33 35 43 61 64 74 86 92

算法特点：每一遍选择出一个最值，确定其在结果序列中的位置，确定元素的位置是从前往后，而每一遍最多进行一次交换，其余元素的相对位置不变。可进行降序和升序排列。

2．冒泡法排序

冒泡法排序的基本思想是：如果有 n 个数，则要进行 $n-1$ 遍的比较，在第一遍的比较中要进行 $n-1$ 次相邻元素的两两比较，在第 j 遍的比较中要进行 $n-j$ 次的两两比较，比较的顺序从前往后，经过一遍的比较后，将最值沉底(换到最后一个元素的位置)，最大值沉底为升序，最小值沉底为降序。

【例 4.8】 有一组整数，采用冒泡法将这 10 个数按从小到大的顺序打印输出。

```
#include <iostream>
using namespace std;
const int n=10;
void main( )
{ int a[n];
  int i,j,x;
```

```
    for(i=1;i<n-1;i++)
        for(j=0;j<n-i;j++)
           if(a[j]>a[j+1]){ x=a[j];a[j]=a[j+1];a[j+1]=x;}
for(int i=0;i<n;i++)cout<<a[i]<<"   ";
cout<<endl;
}
```

从键盘输入如下数据：33　61 43　74　86　92　11　35　64　25

运行结果为：11　25　33　35　43　61　64　74　86　92

算法特点：相邻元素两两比较，每遍将最值沉底，即可确定一个数所在的位置，确定元素所在位置的顺序是从后向前，其余元素可做相应位置调整，可进行升序和降序的排列。

3．插值法排序

插值法排序的基本思想是：将序列分为有序序列和无序序列，依次从无序序列中取出元素值插入到有序序列的合适位置。初始时有序序列中只有一个数，其余 $n-1$ 个数组成无序序列，则 n 个数需进行 $n-1$ 次插入。从无序序列中取出的每个数寻找在有序序列中插入位置时可以从有序序列的最后一个数往前找，在找到插入点之后，可以将插入点位置后的所有元素向后移动一个位置，为插入元素准备空间。

【例 4.9】 用插入法进行排序。

```
#include <iostream>
using namespace std;
void main( )
{ int a[10]={33,61,43,74,86,92,11,35,64,25};
  int i,j,x;
  for(i=1;i<10;i++) //外循环控制遍历次数，n 个数从第 2 个数开始到最后进行 n-1 次插入
  { x=a[i];         //将待插入的数暂存于变量 x 中
      for(j=i-1;j>=0 && x<a[j];j--)     //在有序序列中寻找插入位置
          a[j+1]=a[j];                  //若找到插入位置，则元素后移一个位置
         a[j+1]=x;                      //找到插入位置，完成插入
  }
for(int i=0;i<10;i++)cout<<a[i]<<"   ";
cout<<endl;
}
```

运行结果为：

11 25 33 35 43 61 64 74 86 92

算法特点：该算法的特点是在寻找插入位置的同时完成元素后移，因为元素的移动必须从后向前，所以可将两个操作结合在一起，提高算法效率，该算法可进行升序和降序的排列。

4.4.3　查找

1．顺序查找

顺序查找也称为线性查找，对给定的关键码值 key，从表(数组)的一端开始，依次检查表中每个元素的值是否与给定的关键码值相等，若检查到最后一个元素时仍没有与关键码值相等的元素，则表明未找到。

【例 4.10】 在一组整数 42、55、73、28、48、66、30、65、94、72 中，查找数据为 65 的数据，并给出查找结果。

```
#include <iostream>
using namespace std;
void main( )
{ const int n=10;
int i,a[n]={42,55,73,28,48,66,30,65,94,72};
  int x=65;
for( i=0;i<n;i++)
if(x==a[i]){cout<<"查找"<<x<<"成功，"<<"下标为:"<<i<<endl;break;}
  if (i>=n) cout<<"查找"<<x<<"失败"<<endl;

}
```

运行结果：

查找 65 成功，下标为：7

如果将 x 值改为 88，则运行结果为：

查找 88 失败

2．对分查找

对分查找也称为二分查找，它比顺序查找速度要快，特别是当数据量很大时效果更加明显。首先在有序表中取表的中间位置上的元素 a[mid]=(n–1)/2 与待查元素 key 进行比较，如果 key=a[mid]，则表明找到该元素；如果 key<a[mid]，则应在该表的前一半 a[0]～a[mid–1]中进行查找；如果 key>a[mid]，则在该表的后一半 a[mid+1]～a[n–1]中查找；如此反复进行，直到找到该元素。如果循环结束后 key!=a[mid]，则表明查找失败。

【例 4.11】 假定一维数组中的 n 个元素是一个从小到大顺序排列的有序表，编一程序从数组 a 中用二分查找法，找出其值等于给定值为 key 的元素。

```
#include <iostream>
using namespace std;
const int n=10;
void main( )
{ int a[n]={15,26,37,45,48,52,60,66,73,90};
  int key=80;

int low=0,high=n-1;int mid;
while(low<=high)
  {mid=(low+high)/2;
    if(key==a[mid]) {cout<<"二分查找"<<key<<"成功"<<"下标为"<<mid<<endl;
    break;}
    else if(key<a[mid])high=mid-1;
         else low=mid+1;

  }
if(key!=a[mid])cout<<"二分查找"<<key<<"失败"<<endl;
}
```

该程序的运行结果为：

二分查找 80 失败

如果将 key 的值改为 73，则查找成功。

4.4.4 数组的其他应用

数组在现实生活中的应用还有很多，如对二维表格的计算与处理、矩阵的运算、打印杨晖三角形等。

【例 4.12】 对学生期末考试成绩进行处理，统计每一位学生的平均成绩和每门课程的平均成绩(保留一位小数)，如表 4-1 所示。

表 4-1 学生期末考试成绩表

学号	高等数学	英语	数据结构	宏观经济学	管理学	平均成绩
1	89	81	75	92	88	
2	75	82	80	70	90	
3	88	78	89	90	85	
4	89	80	89	95	96	
…	…	…	…	…	…	
各科平均成绩						

分析：首先需要将各科成绩放入二维数组中，假设一个班有 n 个学生，考试了 m 门课程，可以定义数组 a[n][m]，每个学生平均成绩用 b[n]表示，每门课平均成绩用 c[m]表示。

程序如下：

```
#include <iostream>
#include<iomanip>                  //使用格式函数 setw(n)
using namespace std;
const int n=10,m=5;                //定义 10 人，5 门课
void main( )
{ int a[n][m];
double b[n]={0},c[m]={0};          //初始化平均值变量
  int i,j,sum1,sum2;               //sum1、sum2 分别表示每行的和、每列的和
  for(i=0;i<n;i++)                 //输入学生成绩
      for(j=0;j<m;j++)
          cin>>a[i][j];
for(i=0;i<n;i++)                   //求每行元素和、平均值
{ sum1=0;
   for(j=0;j<m;j++)
      sum1+=a[i][j];
 b[i]=int(sum1/float(m)*10+0.5)/10.0;  //求平均值，保留一位小数，并四舍五入
}
for(j=0;j<m;j++)                   //求每列元素和、平均值
{  sum2=0;
  for(i=0;i<n;i++)
      sum2+=a[j][i];
   c[j]=int(sum2/float(n)*10+0.5)/10.0;
}
```

```
for(i=0;i<n;i++)                    //打印成绩表
{ cout<<setw(15)<<i;
    for(j=0;j<m;j++)
cout<< setw(8)<<a[i][j];
cout<<"平均分:"<<b[i];
cout<<endl;}
cout<<"各科平均分:";
for(i=0;i<m;i++)cout<<c[i]<<setw(8);
cout<<endl;
}
```

【例 4.13】 打印杨辉三角形。

分析：根据杨辉三角形的特点，先将三角形逆时针旋转 30°，变成直角三角形。这样就可以认为杨辉三角是一个由二维数组组成的，数组中只有下三角中有数据，根据数据规律每一行的第一列数字为 1，行数与列数相等位置上的数字为 1。从第三行、第二列开始，每个元素的值为 a[i][j]=a[i−1][j]+a[i−1][j−1]。最后按等腰三角形形式打印。

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
            ……
```

程序如下：

```
#include<iostream>
using namespace std;
  const int n=7;

void main( )
{  int a[n][n],i,j,k;
   for(i=1;i<=7;i++)                //给每行第一列元素和对角线上的元素赋值
    {   a[i][1]=1;
        a[i][i]=1;   }
    for(i=3;i<=7;i++)
         for(j=2;j<=i-1;j++)        //给第 2 列到 i-1 列的元素赋值
             a[i][j]=a[i-1][j]+a[i-1][j-1];
for(i=1;i<=7;i++)                   //输出等腰的杨辉三角形
{ for( k=1;k<=30-2*i;k++)cout<<" ";              //引号中空 1 格
      for(j=1;j<=i;j++)cout<<a[i][j]<<"   ";     //引号中空 2 格
      cout<<endl;
}
}
```

【例 4.14】 求二个矩阵的乘法。

分析：假定矩阵 ***A*** 为 3 行 4 列，矩阵 ***B*** 为 4 行 3 列，根据矩阵乘法的规则，其乘积 ***C*** 为一个 3 行 3 列的矩阵。

```
#include<iostream>
using namespace std;
void main( )
{   int i,j,k;
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
int b[4][3]={12,11,10,9,8,7,6,5,4,3,2,1};
int c[3][3]={0};
  for(i=0;i<3;i++)
      for(j=0;j<3;j++)
      for(k=0;k<4;k++)
        c[i][j]+=a[i][k]*b[k][j];
for(i=0;i<3;i++)
{for(j=0;j<3;j++)
   cout<<c[i][j]<<"  ";
   cout<<endl;
}
}
```

程序运行结果：

50 40
154 128
300 256 216

4.5 字符串

4.5.1 字符串的概念

1．字符串的定义

字符串是用一对双引号括起来的字符序列，例如，"china"、"This is a string."、"中国，北京"都是字符串常量。它在内存中的存放形式是：按串中字符的排列次序顺序存放，每个字符占用 1 字节，每个汉字占用 2 字节。当一个字符串中不含有任何字符时，则称为空串，其长度为 0。在 VC++ 2005 的基本数据类型中，string 为字符串类型的变量，字符个数没有上限，它可以使用可变大小的内存。如果字符串中有特殊字符，需要在特殊字符前加“\”进行转义（原 VC++ 6.0 中是利用一维数组来实现的，并且定义字符数组的长度必须大于等于待存储字符串的长度加 1，且自动在字符串末尾加 “\0”。假设一个字符串的长度为 *n*，则用于存储该字符串的数组的长度至少应为 *n*+1，且每个字符在存储时是按它的 ASCII 码或区位码的形式存储的）。字符串在数组中的表示方法为：

	0	1	2	3	4	5	6	7	8	9	10
字符表示：	s	t	r	i	n	g	s	.	\0		
ASCII 码表示：	83	116	114	105	110	103	115	46	10	0	

2．字符串的输入和输出

能够在定义字符串类型或字符串数组时直接将字符串常量存储到数组中，但不能够通过赋值语句把一个字符串直接赋值给数组变量。

(1) `char a[10]= "china";`

(2) `char b[30]= "He is a student";`

(3) `char c[10]= " ";`

(4) `a="book";`

(5) `a[0]= 'C';`

(6) `string str1,str2;`　　//定义字符串类型的变量

(7) `string str1="He is a student";`

其中第一条语句可改写为：

```
char a[10]={ 'c', 'h', 'i', 'n', 'a', '\0'};
```

最后一个字符\0 是必不可少的，它是字符串在数组中结束的标志。

第二条语句定义了字符数组 b[30]，其中 b[0]～b[14]存储了字符串中的字符，b[15]为'\0'。

第三条语句定义了一个空字符串，它的每个元素的值都是'\0'。

第四条语句为非法的定义。

第五条语句将 a[0]中的字符 c 用 C 来替换。

第六条语句定义了 2 个字符串类型的变量，占用 32 字节的存储空间。

第七条语句将一个字符串赋值给字符串变量 str1。

另外也可以将第一条语句改写成如下的形式:

char a[10]={ 'c', 'h', 'i', 'n', 'a', '0'};　　\\ '\0'可以直接写成常数 0

存储字符串的数组，其元素可以通过下标运算符访问，也可以对它进行整体的输入和输出。假设 a[14]为一个字符串数组，str1 为字符串变量，则:

```
cin>>a>>str1;
cout<<a<<endl;
cout<<str1<<endl;
```

是允许的。当计算机执行第一条语句时，要求用户从键盘上输入一个不含空格的字符串，空格键或回车键被作为字符串输入的结束符，并自动在整个字符串的后面加入一个'\0'。

注意：在向一个字符数组输入字符串时，输入字符串的长度要小于字符数组的长度。输入字符串时不需要加引号。string 是引用类型的变量，其他类型的变量都是值类型的，所以 string 可以指定其值为 null，即字符串变量不引用字符串，且 string 类型的变量在使用前必须初始化。

执行第二条语句时向屏幕输出数组 a 中保存的字符串，并在遇到'\0'时结束。若数组 a 中的内容为：

0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	h	i	n	a	'\0'	B	e	i	j	i	n	g	'\0'

则输出 a 时只会输出"china"，后面的内容不会被输出。

利用一维数组存放字符串，多个字符串的存储就要用二维数组或定义 string 类型的数组变量。例如：

(1) `char a[4][6]={"Bei","jing","shang","hai"};`

(2) `char b[][4]={"sun","mon","tue","wed","thu","fri","sta"};`

(3) `char c[5][10]={" "};`

(4) `string x[]={"sun","mon","tue","wed","thu","fri","sta"};`

第一条语句定义了一个二维字符数组 a，有 4 个字符串，每个串的长度不超过 5 个字符。

第二条语句定义了一个二维字符数组 b，第一维的大小不确定，由后面字符串的个数决定，第二维的大小是 4，说明每个字符串的长度最多为 3 个字符。由于大括号中有 7 个字符串，自动定义 b 数组的第一维下标为 7。

第三条语句定义了一个能存储 5 个字符串的二维字符数组 c，每个字符串的长度不超过 9 个字符，该语句对所有字符串元素初始化为一个空串。

第四条语句定义了能存储 7 个字符串的一维字符串数组。

下面是对二维字符数组的输入和输出举例：

```
char a[5][10];
for(int i=0;i<5;i++)cin>>a[i];//从键盘输入 5 个字符串，每个串的长度不超过 9 个字符
for(i=4;i>=0;i--)cout<<a[i]<<endl;//将数组中的字符串反向输出，并输出一个换行符
```

4.5.2 字符串函数

VC++ 2005 系统抛弃了原来 VC 6.0 中有关 char*字符串处理方法，而采用标准程序库中的 string 类，避免了因内存不足或因字符串长度设置不够引起的问题，使操作更加简便。

注意：这些函数的原型被保存在 string 文件中，当用户在程序中要使用这些函数时，应在程序文件的开始使用#include<string>命令。

1. 声明C++字符串

声明字符串变量方法如下：

```
string str;
```

这里声明了一个字符串变量，其作用是把 str 初始化为一个空字符串。例如:

```
string s;                 //生成一个空字符串 s
string s(str)             //生成 str 的复制品
string s(str, n)          //将字符串 str 内从位置 n 开始的部分当作字符串的初值
string s(str, n, m) //将字符串 str 内从 n 开始的连续 m 个字符赋值给 s
string s(cstr)            //将 cstr 字符串作为 s 的初值
string s(str1, n)         //将 str1 字符串从 n 开始的字符串赋值给 s
string s(num, c)          //生成一个字符串，包含 num 个 c 字符
string s(beg, end)        //以区间 beg;end(不包含 end)内的字符作为字符串 s 的初值
~string()                 //销毁所有字符，释放内存(实际上调用了析构函数)
```

2. 常用的字符串操作函数(详见附录C)

字符串操作函数是 C++字符串的重点，主要函数有：

```
= assign()                //赋以新值
swap()                    //交换两个字符串的内容
+=,append(), push_back()  //在尾部添加字符
insert()                  //插入字符
erase()                   //删除字符
```

clear()　　//删除全部字符
replace()　　//替换字符
+　　//字符串连接运算
==,!=,<,<=,>,>=,compare()　　//字符串比较运算
size(), length()　　//返回字符串长度
max_size()　　//返回字符串的最大个数
empty()　　//判断字符串是否为空
c_str()　　//将内容以 C_string 返回
data()　　//将内容以字符数组形式返回
substr()　　//返回某个子字符串
copy()　　//返回值是复制的字符数
erase()　　//删除函数
replace()　　//替换函数
find()　　//查找函数
substr()　　//求字符串的子串

注意：关于字符串处理的更多函数，请参看其他有关资料或附录 C。

【例 4.15】 观察下列程序的运行结果。

```
#include<iostream>
#include<string>
using namespace std;
void main()
{ string str1(5, 'A');          //将 5 个 'A' 赋值给 str1
   string str2("I am a student.");
   string str3(str2, 7, 8);     //将从第 7 个字符开始的连续 8 个字符赋值给 str3
   cout << str1 << endl;
   cout << str2 << endl;
   cout << str3 << endl;
```

显示结果为：

AAAAA
I am a student.
student.

【例 4.16】 观察下列程序的运行结果。

```
#include<iostream>
#include<string>
using namespace std;
void main()
{   string str2( "I am a student." );
    string s1="my name is Li ming.  ";
      cout << str2 << endl;
      cout << str2+s1 << endl;          // '+' 为字符串连接运算符
    cout<<s1.append("I come from Beijing.")<<endl;
                                        // 将给定的字符串追加到 s1 的后面
```

```
    string s2="wlcome come to ";
    cout<<s2.insert( 15," Bei jing!!")<<endl;
                                        //从第 15 个字符开始插入给定的字符串
}
```

运行结果：

I am a student.

I am a student. my name is Li ming.

my name is Li ming. I come from Beijing.

wlcome come to Bei jing!!

替换函数、删除函数、取子串函数、字符串连接函数、查找函数等也经常使用，例如：

```
string s="il8n";
s.replace(1,2,"nternationalizatio"); //从索引 1 开始的 2 个替换成后面的 C_string
s.erase(13);                         //从索引 13 开始往后全删除
s.erase(7,5);                        //从索引 7 开始往后删 5 个
s.substr( );                         //返回 s 的全部内容
s.substr(11);                        //从 11 往后的子串
s.substr(5,6);                       //从 5 开始连续 6 个字符
find(s,n1,n2) ;
```

返回符合查找条件的字符区间内的第一个字符的索引，没找到目标就返回 npos。这里 s 是被搜寻的对象，n1(可有可无)指出 string 内的搜寻起点索引，n2(可有可无)指出搜寻的字符个数。

4.5.3 字符串应用举例

【例 4.17】 编一程序，首先从键盘上输入一个字符串，接着输入一个字符，然后分别统计字符串中大于、等于、小于该字符的个数。

分析：首先将字符串存入数组 a 中，字符存入 ch 中，统计变量设为 c1、c2、c3。

程序如下：

```
#include<iostream>
using namespace std;
const int n=30;
void main( )
{char a[n];
char ch;
int x1=0;
  int x2;
  int x3;
  x1=x2=x3=0;
cout<<"输入一个字符串";
cin>>a;
cout<<"输入一个字符";
cin >>ch;
int i=0;
while(a[i]){ if(a[i]>ch)x1++;
```

```
  else if(a[i]==ch)x2++; else x3++;i++;}
cout<<"c1="<<x1<<endl;
cout<<"c2="<<x2<<endl;
cout<<"c3="<<x3<<endl;
}
```

【例 4.18】 从键盘输入一个字符串，长度不超过 50，统计出该串中 10 个十进制数字字符的个数并输出。

分析：设字符串放入数组 a 中，要统计的 10 个十进制数字的个数放入数组 b，在未统计前赋初始值 0。借助数字的 ASCII 码进行编程，程序如下：

```
#include<iostream>
using namespace std;
void main( )
{ char a[50];
int x,b[10]={0};
cin>>a;
cout<<endl;
for(int i=0;a[i]!='\0';i++)
if(a[i]>='0'&& a[i]<='9') {x=a[i]-48;b[x]++;}
for(int i=0;i<=9;i++)cout<<i<<"的个数是:"<<b[i]<<endl;
cout<<endl;
}
```

【例 4.19】 编一程序，从键盘输入 10 个字符串，再输入一个待查的字符串，统计出含有待查字符串的个数。

方法一：

```
#include<iostream>
using namespace std;
#include<string.h>
void main( )
{ char a[10][30]={" "};
char ch[30];
int i,n=0;
cout<<"输入 10 个字符串";
for(i=0;i<10;i++)cin>>a[i];
cout<<"输入一个待查字符串";cin>>ch;
for(i=0;i<10;i++)
   if(strcmp(a[i],ch)==0)n++;
   cout<<" 字符串个数:"<<n<<endl;
}
```

方法二：

```
#include<iostream>
using namespace std;
#include<string>
void main( )
```

```
{ string a[10]={" "};
string ch;
int i,n=0;
cout<<"输入 10 个字符串";
for(i=0;i<10;i++)cin>>a[i];
cout<<"输入一个待查字符串";
cin>>ch;
for(i=0;i<10;i++)
   if(a[i]==ch)n++;
   cout<<" 字符串个数:"<<n<<endl;
}
```

本 章 小 结

数组由具有相同数据类型和固定数目的元素组成，占用连续的存储空间。每个数据元素又可用下标变量表示。数组所占用内存空间的大小由数组的类型和数组元素的数目决定。

一维数组用 1 个下标表示。二维数组用 2 个下标表示，其中第一维下标表示行数，第二维下标表示列数。设数组元素个数为 n，则数组的下标为 0～n–1。

字符串占用连续的存储空间，可以用一维数组来存储，默认字符串的最后一个符号为'\0'，所以字符数组空间的大小至少为字符串的个数加 1，每个字符在内存中用 ASCII 码表示。

字符串可以用 string 类型的数据直接赋值和引用。

对字符串的比较、复制、连接等操作是通过字符串函数来实现。

数组在现实生活中用途广泛，如统计、比较、查找、排序、矩阵运算等。

习 题 4

一、选择题

(1) 在 int a[5]={1,3,5,7,9};的定义中，a[3]的值是(　　)。

A. 5　　B. 7　　C. 3　　D. 2

(2) 已知 int a[5];，下列赋值语句错误的是(　　)。

A. a[1]=6;　　B. a[4]=8;　　C. a[5]=10;　　D. a[0]=11;

(3) 已知 string a1="CHINA";，则 cout<<sizeof(a1);语句的输出结果为(　　)。

A. 32　　B. 5　　C. 6　　D. 无限

(4) 已知 int a[4][5];，对数组元素个数的描述正确的是(　　)。

A. 20　　B. 12　　C. 30　　D. 9

(5) 已知 int a[4][5];，对数组元素正确的引用是(　　)。

A. a[3][5]=30;　　B. a[3][−1]=5;　　C. a[4][4]=20;　　D. a[3][4]=30;

二、判断下列描述是否正确，对者打√，错者打×

(1) 在 C++程序中，数组元素的下标是从 0 开始的，数组元素是连续存储在内存单元中的。

(2) 存储在字符数组中的字符串是按其 ASCII 码的形式存放的，每个字符占用一字节，因此数组个数至少要与字符个数相等。

(3) 用 const 定义的整型变量的值可用来开辟数组，但用 int 定义的整型变量不能用来定义数组。

(4) 存储在数组中的字符串的末尾都被系统自动加上了'\0'。

(5) 'A'与"A" 含义是一样的。

三、阅读程序，写出运行结果

(1)

```
#include<iostream>
using namespace std;
#include<string>
void main( )
{   char str[]="hello,world";
    int k=strlen(str);
    char ch;
    for(int i=0; i<k/2; i++)
    {
       ch=str[i];
       str[i]=str[k-i-1];
    str[k-i-1]=ch;
    }
    cout<<str;
}
```

(2)

```
#include<iostream>
using namespace std;
void main( )
{ int x[]={1,3,5,7,9},i;
int *p=x;
for(i=0;i<5;i++)x[i]=*p++;
cout<<x[4];
}
```

(3)

```
#include<iostream>
using namespace std;
#include<string>
char fun(char *c)
{if(*c<='z'&& *c>='a')*c-='a'-'A';
return *c;
}
void main( )
{ char s[100],*p=s;
cin>>s;
```

```
while(*p)
{ *p=fun(p);cout<<*p;p++;}
cout<<endl;
}
```

从键盘输入:

```
Abcdefg↙
```

显示__

如果输入:

```
I am a student
```

显示__

(4)

```
#include<iostream>
using namespace std;
int f(int a[],int n)
{ int i,x;
x=1;
for(i=0;i<=n;i++)x=x*a[i];
return x;
}
void main( )
{ int y,x[]={1,2,3,4,5};
y=f(x,3);
cout<<y<<endl;
}
```

四、编程题

(1) 利用随机函数产生 10 个两位整数，然后按从大到小的顺序排序输出。

(2) 从键盘输入一个不大于 80 个字符的字符串，试分别统计每个英文字母的个数(不区分大小写)。

(3) 设 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 是两个 3×4 的矩阵，写出矩阵 $\boldsymbol{C}=\boldsymbol{A}+\boldsymbol{B}$ 的通用程序。

(4) 定义一个 3×4 的二维数组，从键盘输入各元素的值，求数组中最大元素的值及其所在位置。

(5) 利用随机函数，产生 10 个素数并放入数组 a 中，按逆序输出数组 a 中各元素的值。

第 5 章　指　　针

指针是 C++中的重要概念，它提供了一种较为直观的地址操作的手段。正确合理地使用指针，可以方便、灵活、有效地组织和表示复杂的数据结构。

5.1　指针的概念

在 C++中，任何一种类型的数据都要占用内存中固定个数的存储单元。如 char 型数据(即字符)占用 1 个存储单元；int 型整数占用 4 个存储单元，即 4 字节；double 型实数占用 8 个存储单元，即 8 字节。

指针就是表示地址的一种变量，它的值是内存单元的地址。每个内存单元为二进制的八位，即 1 字节。每个内存单元对应着一个编号(即地址)，计算机控制器(CPU)就是通过这个地址访问(即存取)对应单元中的内容。所以指针值的范围严格来说只能是自然数，并且不能在两个指针间进行加、乘、除这样的运算。由于在 C++中每个数据类型都必有存储空间，所以指针可以应用于几乎所有的数据类型中。所以，从这个角度出发可以将指针分为：指向变量的指针、数组指针、字符指针、指向指针的指针、函数指针、结构变量的指针，以及文件指针等。

数据被存储在一块连续的存储单元中，其第一个单元的地址称为该数据的地址，根据一个数据的地址和该数据的类型就可以存取这个数据。数据的地址称为指向该数据的指针。该数据称为指针所指向的数据。

假定 s 是一个 int 型变量，它的值为 50，定义指针 s 的语句为 int * s，这里 s 就是一个指向整型的指针，可以用于指向一个整型变量。当 p=&s 时，就是将整型变量 s 的地址赋值给 p。当要访问 s 时，实际上是访问 p 所指向的 4 字节内保存的整数 50。指针 p 和 s 之间的关系可用图5-1(a)表示出来，其中矩形框表示为 s 分配的存储空间，带箭头的线段表示“指向”。因为指向 s 的指针 p 也需要存储起来，它占用一个指针数据单元，即 4 字节的存储空间，所以 p 指向 s，也可以用图5-1(b)表示出来。

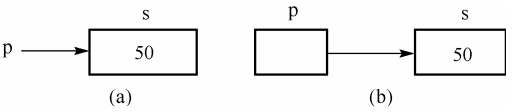


图 5-1　指针与指向数据之间的关系表示

在 C++程序中通常使用指针变量来访问它所指向的数据，此时必须知道它所指向的数据的类型。因为当把一个指针变量定义为指向 int 类型的指针时，它指向的数据将是一个整数；若把一个指针变量定义为指向 double 类型的指针，则它所指向的数据就是一个双精度数。

5.2　指针变量

1．定义格式

在 C++程序中使用指针需要先定义后使用，定义指针的语法形式是:

<类型关键字> *<指针变量名>[=<指针表达式>]，……；

其中<类型关键字>可以是任意类型，指的是指针所指向的对象的类型。

*<指针变量名>表示这里定义了一个指针类型的变量，而不是普通变量。指针变量名前面的类型关健字和星号一起就构成了指针类型。星号字符前、后位置可以不留空格，也可以带有任意多个空格。

需要指出的是，在指针变量定义语句中，可以同时定义多个指针变量，但每个指针变量名前面都必须重写星号字符，每个星号字符同其最前面共用的类型关键字一起构成指针类型。

<指针表达式>是在一个变量名前面加上取地址操作符&。如 a 是一个变量，则&a 就是一个最简单的指针表达式，该表达式的值为存储 a 的数据单元的首地址。把一个变量的地址赋给一个指针变量后，通过这个指针变量就能够间接地存取所指向的变量的值。

定义指针变量语句中的<类型关键字>，除了可以是一般的类型关键字外，还可以是指针类型关键字和无类型关键字(void)。如 int*为 int 指针类型关键字；void 是一个特殊的类型关键字，它只能用来定义指针变量，表示该指针变量无类型。

2．格式举例

```
int *p;                          //定义 p 为一个整型指针变量
int s=10, *ps=&s;                //定义一个整型变量 s 和一个整型指针变量 ps
char x='A', *ch=&x;;             //定义一个字符变量 x 和一个字符指针变量 ch
char *ap="abc",*bp=ab;           //定义字符指针变量 ap 和 bp
void *pl=0,*p2=cp;               //定义两个无类型指针变量 p1、p2
int *dp[20]={0};                 //定义一个整型指针数组 dp[20]，并初始化数组元素为 0
char *ep[3]={"china","Bei","jing"};
//定义一个字符指针数组 ep[3]，其中 ep[0]指向“china”，ep[1]指向“Bei”，ep[2]指向“jing”
int b=10, *xp=&b, **yp=&xp;  //定义一个整型变量 b 等于 10；定义一个整型指针变量
//xp，等于 b 的地址；定义一个整型二级指针变量 yp，等于 xp 的地址
void *pv, i;                     //定义一个 void 类型的指针
int a=10;*na=&a,**np=&na;    //定义一级指针变量 na 和二级指针变量 np
```

一般情况下，指针的值只能赋给相同类型的指针。但 void 类型的指针可以存储任何类型的对象地址，即任何类型的指针都可以赋值给 void 类型的指针变量，经过使用类型强制转换，void 类型的指针便乐意访问任何类型的数据，例如：

```
void *pv,i;
pv=&i;                    //void 类型指针指向整型变量
int *pt=(int *)pv;        //强制类型转换，并将指针所指地址的值赋值给指针变量
```

5.3 指针运算

1．与指针有关的运算符

C++中提供了两个与地址有关的运算符，即“*”和“&”，其中“*”称为指针运算符，表示取指针变量的值；“&”称为取地址运算符，用来得到一个对象的地址。

必须注意“*”和“&”出现在程序中的位置。“*”既可以作为乘法运算符，也可以声明一个变量为指针类型，又可以表示取对象的值。“&”既可以表示声明一个引用变量，也可以表示取对象的地址。例如：

```
int  y=a*b;                 //声明 y 等于 a 乘 b
int *p;                     //声明变量 p 为指针变量
cout<<*p;                   //输出指针变量 p 所指向对象的值
int a,b;
int  &x=a, *pa,*pb=&b;  //声明 x 为 a 的引用，将 b 的地址赋值给 pb
pa=&a;                      //将 a 的地址赋值给 pa
```

2．与指针有关的运算

指针是一种数据类型。与其他数据类型一样，指针变量可以参与部分运算，包括算数运算、关系运算和赋值运算。

像一般的赋值语句一样，指针运算中的赋值语句是将赋值号右边指针表达式的值赋给左边的指针对象，该指针对象必须是一个左值，并且赋值号两边的指针类型必须相同。例如：

```
char  ch='A',*cp;
cp=&ch                          //把 ch 的地址赋给 cp
```

取地址运算符(&)被用在一个指针对象的前面。运算结果是该指针对象的地址。例如：

```
int  a=30, *ap=&a;
cout<<a<<'  '<<ap<<endl;    //输出 a 的值和 a 的地址
```

3．间接访问运算

间接访问操作符(*)，它的后面是一个指针操作数，操作结果是该指针所指的对象的值。例如：

```
int  a=5,b=10;
int *ap=&a,*bp=&b;
cout<<*ap<<'   '<<*bp<<end1;
int  c=*ap+*bp;
*ap+=5;
cout<<*ap<<'   '<<*bp<<'   '<<c<<end1;
```

该程序段的运行结果为：

```
5     10
10    10    15
```

4．加 1 和减 1 运算

其作用是使指针值从当前位置向前或向后移动一个位置。例如：

```
int a[10]={5,7,8,10,12,15,20,15,37,30};   //给数组元素初始化
int  *p=a;          //定义一个整型指针 p，使之指向 a 数组中的第 1 个元素 a[0];
cout<<*p<<'   ';    //输出 a[0]的值
p++;                //指针指向 a[1]
cout<<*p++<<'  ';   //输出 a[1]的值，并使指针指向 a[2]
cout<<*++p<<end1;   //指针向后移动，指向 a[3]，并输出 a[3]的值
```

程序的运行结果是：

```
5   7   10
```

在 C++中，指针不仅可以进行加 1 和减 1 运算，也可以让指针加上或减去一个整数值，其结果是将指针向后或向前移动 n 个数据的地址值。例如：

```
char a[10]="ABCDEF";
char  *p1=a, *p2;
p2=p1+4;
cout<<*p1<<'  '<<*p2<<'  '<<*(p2-1)<<end1;
```

该程序段的运行结果为：

A E D

一个指针也可以减去另一个指针，其值为它们之间的数据个数，若被减数较大则得到正值，否则为负值。例如：

```
int a[10]={5,7,8,10,12,15,20,15,37,30};
int  *p1=a,   *p2=p1+8;
p1++;--p2;
cout<<p2-p1<<'  '<<p1-p2<<end1;
```

该程序段的运行给果为：

6　　–6

5．比较运算

指针是一个地址，地址的大小是后面数据的地址大小，所以两个指针可以比较大小。另外单个指针也可以作为一个逻辑值使用，当它的值不为空时则为逻辑值 true，否则为逻辑值 false。

5.4 指针与数组

指针的加减运算特点，使得指针非常适合于处理存储在一段连续内存空间中同类型的数据。而数组中存放的数据恰好具有这一特点，这样便可以使用指针来对数组及其元素进行方便而快捷的操作。

5.4.1 指针与一维数组

假定 a[n]是一维的 int 类型的数组，该数组占用 4×n 字节的存储空间，该数组的首地址可以通过访问数组名 a 或对 a[0]元素进行取地址运算(&a[0])而得到。如对于数组 a，它的值为 a[0]的地址，是只允许访问而不允许改变的。

一维数组 a[n]中的任何一个数组元素 a[i]的存储地址为 a+i，因为 a[i]是 a 所指向的 a[0]元素之后的第 i 个元素。也可利用间接访问操作符，访问 a[i]元素，其访问表达式为*(a+i)。所以访问数组元素有两种方式，一种是下标方式，另一种是指针方式。

使用下标方式：

```
int a[10]={1,2,3,4,5,6,7,8,9,0};
int  i;
for(i=0;i<10;i++) cout<<a[i]<<'   ';
```

使用指针方式：

```
int a[10]={1,2,3,4,5,6,7,8,9,0};
int  i;
for(i=0;i<10;i++)  cout<<*a+i<<'   ';   //或 cout<<*(a+i)<<'  ';
```

使用指针变量方式：

```
int a[10]={1,2,3,4,5,6,7,8,9,0};
int  i, *p=a;                          //p 指向数组 a 的第一个元素 a[0]
for(i=0;i<10;i++)  cout<<*p++<<'   ';
```

使用指针数组方式：

```
for(i=0;i<10;i++)cout<<p[i]++<<'   ';
```

若指针指向的是字符串，则数组名就是指向字符串的指针，该指针可从第一个字符开始到末尾空字符为止移动。例如：

```
char a[ ]="I am a student";
char *ch=a;                            //a 的值为 char *类型
cout<<ch<<end1;
cout<<ch+5<<end1;
char  b[10];
for (int i=0; i<6; i++)  b[i]=ch[i];
b[i]=0;
cout<<b<<'   '<<&a[6]<<end1;        //&a[6]等同于 a+6
```

该程序段的运行结果为：

```
I am a student
a student
I am a student
```

可以使用 sizeof 运算符对数组名进行运算得到整个数组所占用的存储空间的大小。例如：sizeof(a)计算得到 a 数组所占用存储空间的大小为 15。

【例 5.1】 编写程序，在字符串数组中查找指定的字符，若找到则输出该字符在数组中第一次出现的位置(下标值)，否则输出−1。

```
#include<iostream>
using namespace std;
void main( )
{ char a[]="dfgff1234kjkldssopxyz";
  char *p=a,k;
    cout<<"输入一个字符"<<endl;
  cin>>k;
  for(*p=a[0];*p!='\0';p++)
      if (*p==k)
      {cout<<k<<"位置在"<<p-&a[0]<<"上";break;}
      if (*p=='\0')cout<<-1<<endl;
}
```

5.4.2 指针与二维数组

二维数组在内存中是以行优先的方式按照一维顺序关系存放的。因此对二维数组可以按照一维指针数组来理解，数组名是它的首地址，这个指针数组的元素个数就是行数，每个元素是一个指向二维数组某一行的指针。例如：

```
int a[m][n];              //m 和 n 为已定义的整型常量
```

可以理解为：

a[0]----a[0][0]----a[0][1]----a[0][2]---- ----a[0][n-1]

a[m-1]a[1]----a[1][0]----a[1][[1]----a[1][2]---- ---a[1][n-1]

……

a[m-1]----a[m-1][0]----a[m-1][1]---- ----a[m-1][n-1]

a[i]是该一维数组的数组名，a[i]的值就是指向二维数组 a 中行下标为 i 元素的一维数组的指针，即为 a[i][0]元素的地址，类型为 int*。二维数组 a 中的每个一维元素 a[i]都具有相同的类型，即为一维数组类型 int[n]，每个元素 a[i]的地址(即&a[i])为 int(*)[n]类型，而且 a 的值为 a[0]元素的地址，即二维数组元素 a[0][0]的地址。

由于二维数组 a 为 int 类型，所以指针加 1，即表示指针后移 4 字节，所以 a+i 指向数组 a 的行下标为 i 的一维数组的开始位置，即 a[i][0]元素的位置；对于二维数组 a 中的一维元素 a[i]，其指针访问方式为*(a+i)，所以二维数组 a[m][n]中任一元素 a[i][j]可等价表示为:

(*(a+i))[j] 或 *(*(a+i)+j)或*(a[i]+j)

在二维数组 a[m][n]中，a、a[0]、&a[0]和&a[0][0]的地址值都相同。

【例 5.2】 输出数组元素。

```
#include<iostream>
using namespace std;
void main( )
{int a[2][3]={11,12,13,21,22,23};
int i,j;
int(*p)[3]=a;                  //定义指针变量 p，与 a 的值具有相同的指针类型
for(i=0;i<2;i++)
{ for(j=0;j<3;j++)
    cout<<"  "<<p[i][j];     //采用下标方式访问 p 所指向的二维数组
cout<<endl;  }
}
```

该程序段的运行结果为：

```
11    12    13
21    22    23
```

采用指针访问方式，可将上面的双重循环改为：

```
for(i=0;i<2;i++)
{    int *ch=p[i];             //或使用*p++或使用*(p+1)
   for(j=0;j<3;j++)
       cout<<"   "<<*ch++;  //采用指针访问方式
    cout<<endl;
}
```

也可以改写如下：

```
int   *ch=a[0];          //把 a[0]写成&a[0][0]或(int*)a
 for(i=0;i<2;i++)  {
   for(j=0;j<3;j++)
    cout<<"  "<<*ch++;
 cout<<endl;
}
```

5.4.3　new与delete

1．动态存储分配的概念

通过使用数组，可以对同类型的大量数据进行处理。但在很多情况下，在程序运行之前，并不能确定数组的大小。如果预先将数组开得很大，将造成内存空间的浪费，如果预先开得小了，又影响对数据的处理。因此在 C++中可以使用动态内存分配的方法，即在程序运行阶段从内存的称为“堆存储区”中获得变量的存储空间，每个变量存储空间的大小等于所属类型的长度。当程序运行结束后还可以释放这部分空间。

2．new运算和delete运算

在 C++中建立和删除“堆存储区”使用 new 运算和 delete 运算。当程序执行 new 运算时，将从内存中动态分配的堆存储区内分配一块存储空间，其大小等于 new 运算符后指明的数据类型的长度，然后返回该存储空间的地址，对于数组类型返回的是该空间中存储第一个元素的地址。

new 运算格式：

```
new<数据类型>[(<初值表达式>)]
```

该语句在程序运行过程中动态申请用于存放指定<数据类型>的内存空间，并使用<初值表达式>给出的值进行初始化，其中<初值表达式>是可选项。例如:

```
new double;
```
//动态产生 8 字节的存储空间，并返回一个指向该存储空间的指针。该指针的类型为 double*

```
new int(10);
```
//动态产生 4 字节的存储空间，然后返回一个指向该存储空间的指针，并对该存储空间
//初始化为 10

```
new char[20];
```
//动态产生 20 字节的字符数组空间，然后返回指向该存储空间的首地址的指针，其类型
//为 char*

```
new int[n];
```
//动态产生 n 个整数的数组空间，然后返回一个指向该存储空间首地址的指针，其类型
//为 int*

```
new int[m][n];
```
//动态产生 m×n 个整数的存储空间，然后返回该空间的首地址

使用 new 运算动态分配给变量的存储空间，可以使用 delete 运算重新归还给系统；如果

删除的是对象，则该对象的析构函数将被调用；若没有使用 delete，则只有等到整个程序运行结束才被系统自动收回。

delete 运算格式：

```
delete 指针名；或  delete [ ]指针名;
```

【例 5.3】 动态产生存储 m 个整数的存储空间，并赋值 100 以内的整数。

```
#include<iostream>
using namespace std;
#include<stdlib.h>
#include<time.h>
void main( )
{ srand(time(0));
int i,m;
cout<<"从键盘输入一个整数:";
cin>>m;
int *a=new int[m];
for( i=0;i<m;i++)
{a[i]=rand( )%100;
  cout<<a[i]<<"  ";
}
delete []a;
}
```

5.5 引用变量

引用是变量的别名，当建立引用时，需要用一个变量的名字对它初始化，因此引用将作为其所代表的变量的别名来使用，对引用变量的改动实际上是对其所代表的变量的改动。

引用变量的定义格式：

```
<类型关键字>&<引用变量名>=<已定义的同类型变量>…
```

例如：

```
int i;
  int &ri=i;
```

此处的 ri 是整型变量 i 的引用，即 ri 是 i 的别名。经过这样的说明后，引用 ri 与变量 i 所代表的是同一个变量，占用同一个存储单元。经下面的语句赋值：

```
ri=10;
```

变量 i 的值也为 10。

通常引用类型与它所引用的变量的类型相同，如引用 ri 类型为 int 类型，与 i 的类型相同。

需要说明的是，一旦说明 ri 是 i 的引用，那么在其作用域范围内，引用 ri 不允许再与其他变量建立引用关系。

另外，引用除可以与一般变量建立别名关系外，还可以与使用 new 运算产生的内存变量建立别名关系。例如：

```
float &rf=*new float(15.5);
cout<<rf<<endl;            //输出结果为 15.5
delete &rf;                //释放 rf 所引用的动态内存变量
```

【例 5.4】 引用实例。

```
#include<iostream>
using namespace std;
void main( )
{int i=10,&ri=i,k=20;       //定义 ri 是 i 的引用
ri=k;
cout<<"i="<<i<<"\tri="<<ri<<"\tk="<<k<<endl;
cout<<"&i:"<<&i<<"\t&ri:"<<&ri<<"\t&k:"<<&k<<endl;
                            //在此语句中&作为取地址运算符使用
i++;
cout<<"i="<<i<<"\tri:"<<ri<<"\tk="<<k<<endl;
}
```

运行结果：

```
i = 20    ri = 20    k = 20
&i:0012FF60      &ri:0012FF60      &k:0012FF48
i = 21    ri = 21    k = 20
```

通过对程序运行结果进行分析，可以看出 ri 是 i 的引用，二者所占用的内存空间地址是一样的。由于执行了 ri=k，所以 3 个变量的输出结果一样。执行 i++后，i 与 ri 的值加 1，k 的值不变。

引用变量的几点说明：

(1) 如果程序中出现 int &ri=0;，则表示地址为 0。

(2) &根据出现的位置不同，可以表示为引用或取地址运算符。

(3) 声明引用时，必须同时对其进行初始化。

(4) 引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，且不能再把该引用名作为其他变量名的别名。如 ri=1; 等价于 i=1。

(5) 声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。对引用求地址，就是对目标变量求地址。&ri 与&i 相等。

(6) 不能建立数组的引用。因为数组是一个由若干个元素所组成的集合，所以无法建立数组的别名；

(7) 不能建立引用的引用，不能建立指向引用的指针。因为引用不是一种数据类型，所以没有引用的引用，没有引用的指针。

例如：

```
int n; int &&r=n;
```

是错误的，编译系统把 int &看成一体，把&r 看成一体，即建立了引用的引用。引用的对象应当是某种数据类型的变量。

int &*p=n;也是错误的，编译系统把 int &看成一体，把*p 看成一体，即建立了指向引用的指针。指针只能指向某种数据类型的变量。

(8) 值得一提的是，可以建立指针的引用。

例如：

```
int *p;
int *&q=p;
//正确，编译系统把 int*看成一体，把&q 看成一体，即建立指针 p 的引用，亦即给指针 p 起别名 q
```

本 章 小 结

1. 指针也是一种数据类型，占用 4 字节。除 void 指针类型外，每个指针类型都与一种数据类型相联系，称为指向该数据类型的指针。

2. 一个变量的地址可以用&运算符计算得到，也可以将它赋给同类型的指针变量，并通过*间接访问运算得到指针所指变量的值。

3. 当指针指向一个数组时，它的值为第一个元素的地址，不允许给数组名赋值。

4. 可以用指针访问数组元素，其访问方式可以采用下标方式，也可以采用指针方式。

5. 使用 new 运算符能够进行动态存储空间分配，特别适用于数组空间大小事先不确定的情况，可以采用 delete 运算符释放动态分配的存储空间。

习 题 5

一、选择题

(1) 设有说明语句 int i=2, *p=&i ; char s[20]="hello", *q=s; 以下选项中存在语法错误的是(　　)。

A. cin>>p;　　B. cout<<p;　　C. cin>>q;　　D. cout<<q;

(2) 设有说明语句 char a[ ]="string!" , *p=a; 以下说法正确的是(　　)。

A. sizeof(a)的值与 sizeof(p)的值相等　　B. sizeof(a)的值与 sizeof(*p)的值相等

C. sizeof(a)的值与 sizeof(a[0])的值相等　　D. 上述说法都是错的

(3) 指向同一个数组的两个指针，做(　　)运算是没有意义的。

A. 加法　　B. 减法　　C. 比较　　D. 赋值

(4) 已知 int a=10; int &ra=a; 当执行++a;后，引用变量 ra 的值为(　　)。

A. 10　　B. 11　　C. 12　　D. 不确定

(5) 下列关于动态分配内存的说法，错误的是(　　)。

A. new 和 delete 是 C++语言提供的运算符

B. delete 只能释放 new 分配的内存空间

C. 用 new 分配的内存空间，其数量可以用常量表示，也可以用变量表示

D. 使用 new 和 delete 需要加入头文件<string>

二、判断题

(1) 一个指针只能指向与它类型相同的变量。

(2) 指针是变量，引用不是变量。

(3) 指针和引用在创建时都必须初始化。

(4) char const * p;该语句的含义是：指针所指内容不可改，即*p 是常量字符串。

(5) 一个指针类型的对象占用内存的 4 字节的内存空间。

(6) 空类型指针不能进行指针运算，也不能进行间接引用。

(7) 使用取地址操作符，可以取得指针变量自身的地址，但取不到引用变量自身的地址。

三、填空题

(1) 设 p 是一个指向整型变量的指针，则用_____表示该整型变量的值，用_____表示指针变量的地址。

(2) 设 p 是一个指针变量，则*p++运算的结果是_____。

(3) 设 p 所指对象的值为 25，p+1 所指对象的值为 45，则*p++的值为_____。

(4) 设 a 是一个一维数组，则 a[i]的指针访问方式是_____。

(5) 若 p 指向 x，则_____与 x 的表示是等价的。

(6) int a[]={5,4,3,2,1},*p[]={a+3,a+2,a+1,a},**q=p，表达式*(p[0]+1)+**(q+2)的值是_____。

四、阅读程序，写出执行结果

(1)

```
#include<iostream>
using namespace std;
void main()
{ int a[10]={0},*p;
  p=a;
  cout<<p<<"  "<<a<<endl;
  cout<<*p<<"  "<<a[0]<<endl;
  cout<<sizeof(*p)<<"  "<<sizeof(p)<<"  "<<sizeof(a)<<endl;
  }
```

(2)

```
#include<iostream>
using namespace std;
void main()
{ int i,a[8]={14,16,10,12,11,39,13,25};
  int *p=a;
  for(i=0;i<8;i++)
  {cout<<"    "<<*p++;
  if((i+1)%4==0)cout<<endl;
  }
}
```

(3)

```
#include<iostream>
using namespace std;
const int n=5;
void main()
{int a[n]={3,10,6,8,7};
int *p1=a,*p2=a+n-1;
while(p1<p2)
{ int x=*p1;*p1=*p2;*p2=x;
  p1++;p2--;}
for(int i=0;i<n;i++)
```

```
cout<<*(a+i)<<"  ";
cout<<endl;
 }
```

(4)

```
#include<iostream>
using namespace std;
const int n=12;
void main( )
{ int *a=new int[n];
a[0]=0;a[1]=1;
for( int i=2;i<n;i++)
a[i]=a[i-1]+a[i-2];
for(int i=0;i<n;i++)
{cout<<"  "<<a[i];
if((i+1)%5==0)cout<<endl;
}
delete []a;
}
```

(5)

```
#include<iostream>
using namespace std;
void main( )
{ int a[8]={21,34,65,77,37,98,20,35};
  int s=0;
  int *p=a+3;
  while(p<a+8)s+=*p++;
  cout<<s<<' '<<float(s)/5<<endl;
}
```

五、编程题

(1) 从键盘输入一个字符串，统计该串中所有十进制数字字符的个数。

(2) 用指针方法编程，将一个 3×4 的矩阵进行转置。

(3) 利用随机函数产生 10 个两位正整数，然后分别统计出偶数和奇数的和。

(4) 用指针方法编程，查找一个指定字符在某一字符串中第一次出现的位置，若找到，则输出其位置号，否则输出−1。

第 6 章　函　　数

函数是一个能完成某一独立功能的子程序，也叫做程序模块。函数是对复杂问题的一种“自顶向下，逐步求精”思想的体现。程序员可以将一个大而复杂的程序分解为若干个相对独立且功能单一的小块程序(函数)进行编写，并通过在各个函数之间进行调用，实现总体的功能。

设计 C++程序的过程，实际上就是编写函数的过程，至少要编写一个 main()函数。

执行 C++程序，就是执行相应的 main()函数。即从 main()函数的第一个左花括号“{”开始，依次执行后面的语句，直到最后一个右花括号“}”为止。如果在执行过程中遇到其他的函数，则调用其他函数。调用完后，返回到刚才调用函数的下一条语句继续执行。而其他函数也只有在执行 main()函数的过程中被调用时才会执行。

函数可以被一个函数调用，也可以调用另一个函数，它们之间可以存在着调用上的嵌套关系。但是，C++不允许函数的定义嵌套，即在函数定义中再定义一个函数是非法的。

6.1　函数的定义与调用

C++程序中在调用函数之前，首先要对函数进行定义。如果调用此函数在前，函数定义在后，就会产生编译错误。

为了使函数的调用不受函数定义位置的影响，可以在调用函数前进行函数的声明。这样，不管函数是在哪里定义的，只要在调用前进行函数的声明，就可以保证函数调用的合法性。

6.1.1　函数的定义

函数定义的一般格式为：

[<有效范围>]<类型名><函数名><参数表><函数体>

<有效范围>由所使用的保留字 extern 或 static 决定。extern，表示定义一个全局函数或外部函数，它在整个程序的所有程序文件中都有效，即能够被任何程序文件调用。static，表示定义一个局部函数或静态函数，它只在所属的程序文件中有效，即只能被所属的程序文件调用。若省略<有效范围>选项，则默认为使用保留字 extern。

<类型名>是该函数的类型，也就是该函数返回值的类型，此类型可以是 C++中除函数、数组类型之外的任何一个合法的数据类型，包括普通类型、指针类型和引用类型等。

函数的返回值通常指明函数处理的结果，由函数体中的 return 语句给出。一个函数可以有返回值，也可以无返回值(称为无返回值函数或无类型函数)。此时需要使用保留字 void 作为类型名，而且函数体中也不需要再写 return 语句，或者 return 的后面什么也没有。每个函数都有类型，如果在函数定义时没有明确指定类型，则默认类型为 int。

<函数名>是一个有效的 C++标识符，遵循一般的命名规则。在函数名后面必须跟一对小括号“()”，用来将函数名与变量名或其他用户自定义的标识符区分开来。在小括号中可以没

有任何信息，也可以包含形式参数表。C++程序通过使用这个函数名和实参表来调用该函数。主函数的名称规定取编译器默认的名称 main()。

<参数表>又称形式参数表，写在函数名后面的一对圆括号内。它可包含任意多个(含 0 个，即没有)参数说明项，当多于一个时其前后两个参数说明项之间必须用逗号分开。

每个参数说明项由一种已定义的数据类型和一个变量标识符组成，该变量标识符称为该函数的形式参数，简称形参，形参前面给出的数据类型称为该形参的类型。每个形参的类型可以为任一种数据类型，包括普通类型、指针类型、数组类型、引用类型等。

函数定义中的<参数表>可以被省略，表明该函数为无参函数。若<参数表>用 void 取代，则也表明是无参函数。若<参数表>不为空，同时又不是保留字 void，则称为带参函数。

<函数体>是一条复合语句，它以左花括号开始、右花括号结束，中间为一条或若干条 C++语句，用于实现函数执行的功能。

注意：在一个函数体内允许有一个或多个 return 语句，一旦执行到其中某一个 return 语句时，return 后面的语句就不再执行，直接返回调用位置继续向下执行。

例如：

```
int max(int a,int b)
{
   int t;
   if(a>b) t=a;
   else t=b;
   return t;
}
```

该函数的功能是返回 a 和 b 中的大者。

函数形参也可以在函数体外说明。例如：

```
func1(int a, int b)
{
    …
}
```

也可写成：

```
func1(a,b)
int a;
int b;
{
…
}
```

【例 6.1】 观察下列程序的运行结果。

```
#include<iostream>
using namespace std;int func(int n)
{if(n>0)
  return 1;
  else if(n==0)
  return 0;
```

```
  else return -1;
  }
void main( )
  {
  int n;
  cout<<"Please input n:"<<endl;
  cin>>n;
  cout<<"\nthe result:"<<func(n)<<endl;
  }
```

此程序的运行结果为：

```
Please input n:
5
the result:1
Please input n:
–5
the result:–1
```

注意：C++中不允许函数定义嵌套，即在函数定义中再定义一个函数是非法的。一个函数只能定义在别的函数的外部，函数定义之间都是平行的，互相独立的。

例如，下面的代码在主函数中非法嵌套了一个 a()函数定义：

```
void main( )
{
  void a( )
  {
    函数体;
  }
}
```

6.1.2　函数的声明与调用

在函数定义中，函数体之前的所有部分称为函数头，它给出函数名、返回类型、每个参数的次序和类型等函数原型信息，所以当没有专门给出函数原型声明语句时，系统就从每个函数头中获取函数原型信息。

函数必须先定义或声明后才能被调用，否则编译程序无法判断该调用的正确性。在一个完整的程序中，函数的定义和函数的调用可以在同一个程序文件中，也可以放在不同的程序文件中，但必须确保函数原型语句与函数调用表达式出现在同一个文件中，且函数原型语句出现在前，函数调用出现在后。因此函数原型声明，就是告诉编译器函数的返回类型、名称和形参表构成，以便编译系统对函数的调用进行检查。

函数声明的一般格式为：

```
函数类型 函数名(形式参数表);
```

除了需在函数声明的末尾加上一个分号“;”之外，其他的内容与函数定义中的第一行(称函数头)的内容一样。

例如，设有一函数的定义为：

```
double f1(int a, int b, double c)
{
函数体
}
```

正确完整的函数原型声明应为：

```
double f1(int x, int y, double z); //末尾要加上分号
```

也可以写为如下形式：

```
double f1(int, int, double);        //函数声明中省略了形参名
```

下面形式的写法是错误的：

```
double f1(x,y,z);                   //函数声明中省略了形参类型
```

或：

```
f1(int x, int y, double z);         //函数声明中省略了函数类型
```

或：

```
double f1(int x, double z, int y); //函数声明中形参顺序调换了
```

在 C++中，除了主函数 main 由系统自动调用外，其他函数都是由主函数直接或间接调用的。函数调用的语法格式为：

```
函数名(实际参数表);
```

实际参数表中每个表达式称为实参，每个实参的类型必须与相应的形参类型相同或兼容。每个实参是一个表达式，包括常量、变量、函数调用表达式，或带运算符的能确切计算出一个值的表达式。例如：

```
f1(25);              //实参是一个整数
f2(x);               //实参是一个变量
f3(a,2*a+4) ;        //第一个是变量，第二个是运算表达式
f4(sin(x),'a');      //第一个是函数调用表达式，第二个是字符常量
f5(&a,*p, a/b+4) ;  //分别为取地址运算、间接访问和一般运算表达式
```

常见的函数调用方式有下列两种：

方式 1：将函数调用作为一条表达式语句使用，只要求函数完成一定的操作，而不使用其返回值。若函数调用带有返回值，则这个值将会自动丢失。例如：

```
max(3,5);
```

方式 2：对于具有返回值的函数来说，把函数调用语句看做语句一部分，使用函数的返回值参与相应的运算或执行相应的操作。例如：

```
int a=max(5,7);
int a=max(5,7)+1;
cout<<max(5,7)<<endl;
if(f1(a,b)) cout<<"true"<<endl;
int a=2; a=max(max(a,5),7);
```

【例 6.2】 求最大值函数。

```
#include<iostream>
using namespace std;
int max(int a,int b,int c);
void main( )
{
  int x,y,z;
  cout<<"Please input x y z:"<<endl;
  cin>>x>>y>>z;
  int m=max(x,y,z);
  cout<<"The max is:"<<m<<endl;
}
int max(int a,int b,int c)
{
  int t;
  t=a;
  if(b>t) t=b;
  if(c>t) t=c;
  return t;
}
```

6.2　函数调用方式和参数传递

函数调用是实现函数功能的手段，C++中的函数调用包括传值调用和传址调用。

6.2.1　函数调用过程

调用函数分三步：第一步是参数传递，第二步是执行函数体，第三步是返回，即返回到函数调用表达式的位置。因此在函数体中一般都有一个 return 语句，但当函数为 void 类型时，称为无返回值函数，此时 return 语句可以省略。

参数传递称为“实虚结合”，即实参向形参传递信息，使形参具有确切的含义(即具有对应的存储空间和初值)。这种传递又分为两种不同的方式，一种是按值传递，另一种是按址传递和引用传递。

6.2.2　传值调用

使用传值调用时，调用函数的实参用常量、变量或表达式，被调用的函数的形参用变量。

调用时系统先计算表达式的值，再将实参的值按位置对应地赋值给形参，即对形参进行初始化。形参得到值后，在函数运算中其值可以改变，但这只影响到函数体中的形参值，对实参没有影响。所以，传值调用的特点是形参的变化不影响实参。

【例 6.3】传值调用的例子。

```
#include<iostream>
using namespace std;
void swap(int,int);
void main( )
```

```
{
  int a=3,b=4;
  cout<<"a="<<a<<",b="<<b<<endl;
  swap(a,b);              //函数调用
  cout<<"a="<<a<<",b="<<b <<endl;
}
void swap(int x,int y)  //无返回值函数
{
  int z=x;
  x=y;
  y=z;
}
```

此程序的运行结果为：

a = 3, b = 4

a = 3, b = 4

从该程序的运行结果看，函数的功能是实现两数交换，但函数调用前后 a、b 的输出结果是一样的，可见虚参的变化不影响实参。

如果想让实参也发生改变，就不能用传值调用，而应改为传址调用或引用调用。

6.2.3 传址调用

使用传址调用时，调用函数的实参用地址值，被调用的函数形参用指针。函数调用时，系统将把实参的存储地址传送给对应的形参指针，从而使得形参指针指向实参地址。因此，被调用函数中对形参指针所指向的地址中内容的改变会影响到实参。即传址调用的特点是可以通过改变形参所指向变量的值影响到实参。

【例 6.4】 传址调用的例子。

```
#include<iostream>
using namespace std;
void swap(int *,int *);
void main( )
{int a=3,b=4;
 cout<<"a="<<a<<",b="<<b<<endl;
 swap(&a,&b);                    //函数调用，实参的地址传给形参
 cout<<"a="<<a<<",b="<<b<<endl;
}
void swap(int *x,int *y)        //形参用指针变量
{
  int z=*x;
  *x=*y;
  *y=z;
}
```

此程序的运行结果为：

a = 3, b = 4

a = 4, b = 3

从该程序的运行结果来看，函数的功能将 x 和 y 做了交换，程序运行结果也将 a 和 b 的值做了交换。可见，传址调用可以在被调用的函数中改变形参的值后，实参的值也相应发生了变化。但如果在函数中反复利用指针进行间接访问，会使程序容易产生错误且难以阅读。如果改为引用调用的方式，则既可以使得对形参的任何改变影响到实参，又使函数调用显得更方便、更直接。因此在 C++中经常使用引用作为函数的形参。引用调用是在形参前面加上引用运算符“&”。

【例 6.5】 引用调用的例子。

```
#include<iostream>
using namespace std;
void swap(int &,int &);
void main( )
{
  int a=3,b=4;
  cout<<"a="<<a<<",b="<<b<<endl;
  swap(a,b);                    //函数调用，实参用变量的方式
  cout<<"a="<<a<<",b=" <<b<<endl;
}
void swap(int &x,int &y)      //形参用引用的方式
{
  int z=x;
  x=y;
  y=z;
}
```

此程序的运行结果为：

a = 3, b = 4

a = 4, b = 3

从该程序的运行结果可以看出，在引用调用方式中，实参用变量名，形参用引用，调用时将实参的值赋值给形参的引用变量。程序执行后，引用变量的值发生变化，对应的实参的值也发生变化。在 C++编程中，经常使用传值和引用的方法，较少使用传址的方法，因为传址调用要用到指针，而用指针传递参数容易出错。

6.2.4　数组作为参数调用

数组作函数的参数时，实质上是将实参中数组的首地址传递给形参。当调用函数的实参用数组名，被调用函数的形参用数组，这种调用机制是形参和实参共用内存中的同一块区域，因此函数中数组值的改变也会影响到实参中数组元素的值。

【例 6.6】 分析下面的程序。

```
#include<iostream>
using namespace std;
int a[10]={12,34,45,3,8,33,67,14,48,74};
void sort1(int b[],int n);
void main( )
{int i;
```

```
  for(i=0;i<10;i++)          //输出原数组元素的值
      cout<<a[i]<<"  ";
  cout<<endl;
  sort1(a,10);               //调用函数
  for(i=0;i<10;i++)          //输出排序后的数组元素值
  cout<<a[i]<<"  ";
  cout<<endl;
}
void sort1(int b[],int n)
{ int i,j,k;
  for(i=0;i<9;i++)
      for(j=i+1;j<10;j++)
        if(b[i]<b[j])
          { k=b[i];b[i]=b[j];b[j]=k;}
        return;
}
```

程序运行结果：

```
12  34  45  3  8  33  67  14  48  74
74  67  48  45  34  33  14  12  8  3
```

该程序中，开始定义了一个外部一维数组 a，还定义了一个排序函数。函数中形参使用b[]数组，主函数 main()中调用语句的实参使用数组名 a。从输出结果看，调用函数后数组 a 中元素发生了变化，说明数组 a 和 b[]占用同一存储单元。

在 C++中，数组名被规定为是一个指针，该指针指向该数组元素的首地址，因此数组名就是一个常量指针。实际上，形参用指针，实参用数组名，也是可以的。

【例 6.7】 分析下面的程序。

```
#include<iostream>
using namespace std;
int a[10]={12,34,45,3,8,33,67,14,48,74};
void sort1(int *p,int n);
void main( )
{int i;
  for(i=0;i<10;i++)
      cout<<a[i]<<"  ";
  cout<<endl;
  sort1(a,10);                    //实参用数组名
  for(i=0;i<10;i++)
      cout<<a[i]<<"  ";
  cout<<endl;
}
void sort1(int *p,int n)          //形参用指向数组的指针
{ int i,j,k;
  for(i=0;i<9;i++)
      for(j=i+1;j<10;j++)
         if(p[i]<p[j])
```

```
            { k=p[i];p[i]=p[j];p[j]=k;}
          return;
}
```

另外，对实参用数组名，形参用引用方式，也是可以的。这里先用类型定义语句定义一个 int 类型的数组类型，例如：

```
typedef int array[10];
```

然后使用 array 来定义数组和引用。

【例 6.8】 分析下面的程序。

```
#include<iostream>
using namespace std;
typedef int array[10];
int a[10]={12,34,45,3,8,33,67,14,48,74};
void sort1(array &b,int n);
void main( )
{int i;
 for(i=0;i<10;i++)
     cout<<a[i]<<"  ";
 cout<<endl;
 sort1(a,10);                //使用数组名调用函数
 for(i=0;i<10;i++)
     cout<<a[i]<<"  ";
 cout<<endl;
}
void sort1(array &b,int n)  //使用引用作为参数
{ int i,j,k;
 for(i=0;i<9;i++)
     for(j=i+1;j<10;j++)
         if(b[i]<b[j])
          { k=b[i];b[i]=b[j];b[j]=k;}
         return;
}
```

程序运行结果：

```
12  34  45  3  8  33  67  14  48  74
74  67  48  45  34  33  14  12  8  3
```

6.3　变量的作用域

变量的作用域用于定义变量可以在哪些代码区域中起作用，类似于命名空间。这种机制允许用于在不同的环境下使用相同命名的对象，这些对象不会因为命名相同而造成冲突。变量的作用域又称作用范围，每个变量都遵从先定义后使用的原则。根据变量在程序中出现的位置不同，其作用域范围也不同。一个变量离开了它的作用域范围，系统将自动释放其占用的内存空间，因此该变量也就不可见了。

6.3.1 作用域分类

变量的作用域按其作用域的大小分为四个层次：全局作用域、文件作用域、函数作用域和块作用域。

全局作用域：其变量的作用范围最大，它包含着组成该程序的所有文件。当一个变量的定义语句出现在一个程序文件的所有函数之外，并且不带有任何存储属性标识符或使用 extern 属性标识符时，则该语句定义的变量具有全局作用域，它在整个程序的所有文件中都有效，即在所有文件中都是可见的。当一个全局变量没有在本程序文件中定义，但在本程序中使用，则必须在本程序文件开始进行声明，例如:

```
extern  <类型名><变量名表>
```

其中，<变量名表>可以包含多个变量，各变量之间用逗号分割，若在定义时没有初始化，则系统默认的初始值为 0。

文件作用域：具有文件作用域的变量仅在定义它的文件内有效。当一个变量定义语句出现在程序文件中的所有函数定义之外，且该语句带有 static 存储属性时，则该语句定义的所有变量都具有文件作用域，该变量在其他文件中无效、不可见的。若在定义文件作用域变量时没有初始化，则默认的初始值为 0。另外，宏定义所定义的符号常量一般具有文件作用域，在没有使用 undef 命令之前，其作用域从定义时起，到该文件结束时止。

函数作用域：一般函数的形参和在函数中定义的自动类变量和内部静态类变量，以及函数中定义的语句标号都属于具有函数作用域的变量。但由于语句标号不是变量，因此语句标号不属于变量的作用域，标号仅在定义它的函数内有效。

块作用域：当一个变量是在一个函数体内定义时，则称它具有块作用域，其作用域范围是从定义点开始，直到该块结束(即所在复合语句的右花括号)为止。

具有块作用域的变量称为局部变量，若局部变量没有被初始化，则系统不会对它初始化，它的初值是不确定的。对于在函数体中使用的变量定义语句，若在其前面加上 static 保留字，则称所定义的变量为静态局部变量，若静态局部变量没有被初始化，则编译时会被自动初始化为 0。

对于非静态局部变量，每次执行到它的定义语句时，都会为它分配对应的存储空间，并对带初值表达式的变量进行初始化；而对于静态局部变量，只是在整个程序执行过程中第一次执行到它的定义语句时为其分配对应的存储空间，并进行初始化，以后再执行到它时什么都不会做，相当于第一次执行后就删除了该语句。

函数中定义形参具有块作用域，这个块是作为函数体的复合语句，当离开函数体后它就不存在了，函数调用时为它分配的存储空间也就被系统自动回收了，当然引用参数对应的存储空间不会被回收。由于每个形参具有块作用域，所以它也是局部变量。

在 C++程序中定义的符号常量也同变量一样具有不同的作用域。当符号常量定义语句出现在所有函数定义之外，并且在前面带有 extern 保留字时，则所定义的常量具有全局作用域。若在前面带有 static 关键字或什么都没有，则所定义的常量具有文件作用域。若符号常量定义语句出现在一个函数体内，则定义的符号常量具有局部作用域。

利用 new 运算符动态分配的对象，它的作用域和生存期都是从动态分配建立对象开始到采用 delete 运算符回收或整个程序结束为止。

具有同一作用域的任何标识符，不管它表示什么对象(如常量、变量、函数、类型等)，都不允许重名，若重名系统就无法唯一确定它的含义了。

由于每个复合语句就是一个块，所以在不同复合语句中定义的对象具有不同的块作用域，也称为具有不同的作用域，其对象名允许重名，因为系统能够区分它们。

6.3.2　应用举例

【例 6.9】 作用域举例。

```
#include<iostream>
using namespace std;
void add(int num);      //文件作用域
void change_sum( );     //文件作用域
int sum=1;              //文件作用域
int main( )
{   int num = 5;        //在主文件中有效，具有局部作用域，仅在该大括号内有效
       add(num);        //调用函数 add
       cout<<"main num="<<num<<endl;        //输出结果为 5
       for(int sum=0, num=0;num<5;num++)
                        //此处的 sum 和 num 具有块作用域，仅在 for 循环内有效
       {     sum+=num;
             cout<<"num="<<num;             //输出 0～5
              cout<<"for sum="<<sum<<endl;  //输出 0～5 的叠加和
       }
       {      int i;    //i 具有块作用域，仅在该大括号内可见
              for(i=0;i<10;i++);            //注意循环语句后的分号
              cout<<"i="<<i<<endl;          //输出结果为 10
       }
      cout<<"main sum="<<sum<<endl;         //输出 1
      cout<<"main num="<<num<<endl;         //输出 5
       change_sum( );
       cout<<"file sum="<<sum<<endl;        //输出全局的 sum，结果为 2
       return 0;
}
void add(int num)   //文件作用域
{       num++;      //num 具有局部作用域
       cout<<"add num="<<num<<endl; }
void change_sum( )  //文件作用域
{       sum++;      //sum 具有文件作用域
       cout<<"change_sum="<<sum<<endl;
}
```

程序的运行结果为：

```
add num = 6
main num = 5
num = 0for sum = 0
num = 1for sum = 1
```

num = 2for sum = 3
num = 3for sum = 6
num = 4for sum = 10
i = 10
main sum = 1
main num = 5
change_sum = 2
file sum = 2

从本例中可以清楚地明白局部作用域和文件作用域的概念。另外注意文件作用域不仅限于变量，也包括函数。在文件作用域中函数也是以其声明开始，到文件结尾结束。而且当拥有文件作用域与拥有局部作用域变量同名时，不会发生冲突。

【例 6.10】 在块作用域内引用文件作用域的同名变量。

```
#include<iostream>
using namespace std;
int i=10;
void  main()
{   {
        int i=5,k;
        ::i=::i+4;
        k=::i+i;
        cout<<"i="<<i<<",::i="<<::i<<",k="<<k<<endl;
    }
    cout<<"::i="<<i<<endl;
}
```

程序的运行结果为：

i = 5, ::i = 14, k = 19
::i = 14

从程序运行结果可以看出，在函数外部定义的标识符或用 extern 说明的标识符都为全局变量，其作用域为文件作用域，它从声明之处开始，到文件结束一直是可见的。当块作用域内的局部变量与全局变量同名时，局部变量优先，但可在块作用域内使用作用域标识符“::”来引用与局部变量同名的全局变量名。

【例 6.11】 带默认形参值的函数调用。

在 C++语言中允许在函数的说明或定义时给一个或多个形参指定默认值。当一个函数既有定义又有声明时，形参的默认值必须在声明中指定，而不能放在定义中指定。只有当函数没有声明时，才可以在函数定义中指定形参的默认值。默认值的定义必须遵守从右到左的顺序。即如果某个形参没有默认值，则它左边的参数就不能有默认值。例如：

```
int add(int a, int b,int c=3);      //合法
int add(int a=1, int b, int c=3);   //不合法
```

在函数调用时，编译器按从左到右的顺序将实参与形参结合，当实参的数目不足时，编译器将按同样的顺序用说明或定义中的默认值来补足所缺少的实参。例如：

```
#include<iostream>
using namespace std;
```

```
int m=10;
int add(int x,int y=7,int z=m);
void  main( )
{
  int a=5,b=15,c=20;
  int sum=add(a,b);
  cout<<"sum="<<sum<<endl;
}
int add(int x,int y,int z)
{ return x+y+z;}
```

程序的运行结果为：

sum = 30

从运行结果可以看出，x 取值为 5，y 取值为 15，z 取值为 m 的值 10。即在函数调用时没有给定 z 的值，因此取默认值 m=10。

【例 6.12】 计算 1+2+3+ … +n 前 n 个整数之和。

```
#include<iostream>
using namespace std;
int add(int n)
{static int sum=0;
sum+=n;
return sum;
}
static int k;
void main( )
{ int i;
for(i=1;i<=5;i++)
{k=add(i);
cout<<"前"<<i<<"个整数之和为:"<<k<<endl;
}
}
```

程序运行结果：

前 1 个整数之和为:1
前 2 个整数之和为:2
前 3 个整数之和为:6
前 4 个整数之和为:10
前 5 个整数之和为:15

6.4　递归函数

1．函数的嵌套调用

在 C++中，一个函数可以调用另一个函数，这个被调用函数还可以调用其他函数，并形成任何深度的调用层次。例如：

```
int fun1(int a, float b)
   { int  c;
  c=fun2(a,a+b);
   return c;
}
int fun2(int x, int y)
   {int z;
    z=fun3(int n);
    return z;
   }
int fun3(int n)
{ int p=0,k;
   for(k=0;k<n;k++)
   p+=k;
   return p;
}
```

这里 fun1、fun2 和 fun3 都是独立定义的函数。

2．函数的递归调用

在调用一个函数的过程中，函数的某些语句又直接或间接地调用函数本身，这就形成了函数的递归调用。递归调用有两种方式：直接递归调用和间接递归调用。直接递归调用即在一个函数中调用自身；间接递归调用即在一个函数中调用了其他函数，而在该其他函数中又调用了本函数。图 6-1(a)表示了直接递归调用，图(b)表示了间接递归调用:

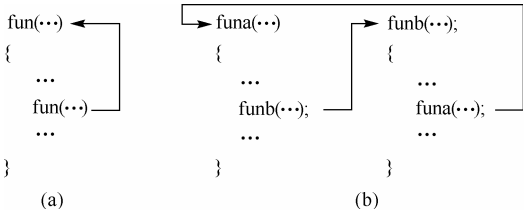


图 6-1　函数的递归调用

在 fun 内部的某条语句调用了 fun 函数本身，构成了直接递归调用。

funa 函数内部的某条语句调用了 funb，而 funb 函数的某条语句又调用了 funa，构成了间接递归调用。

不论是直接递归调用还是间接递归调用，递归调用都形成了调用的回路，如果递归的过程没有一定的中止条件，程序就会陷入类似死循环的情况。因此，在设计递归函数时，可以使用分支语句进行控制，一定要保证递归过程在某种条件下可以结束。下面通过一个例子来解释递归的过程。

【例 6.13】 求 n 的阶乘 n!。

```
#include<iostream>
using namespace std;
long fact(int n)
{if(n<0)
 {cout<<"error!"<<endl;
  return -1;
```

```
}
  else if(n==1)return 1;
else return n*fact(n-1);
}
void main( )
{long fact(int n);
  int n;
  cout<<"please input n:" <<endl;
cin>>n;
  cout<<"n!="<<fact(n)<<endl;}
```

此程序的运行结果为：

please input n:

输入 3，回车后得到结果如下：

n!=6

分析程序的运行过程：

（1）程序开始运行后，进入 main 函数，通过调用 fact 函数，输入变量 n(假设输入 3)。

（2）开始调用函数 fact(n)，这里传递实际参数 3。

（3）进入函数 fact(3)，3!=3*2!，因此准备执行 fact=3*fact(2)，在计算 fact 之前，先调用 fact(2)。

（4）进入函数 fact(2)，2!=2*1!，因此准备执行 fact=2*f(1)，在计算 fact 之前，先调用 fact(1)。

（5）进入函数 fact(1)，1!=1，因此 fact=1，本次函数调用结束并返回 1。

（6）回到 fact(2)；计算 fact=2*1，fact(2)调用结束并返回 2。

（7）回到 fact(3)；计算 fact=3*2，fact(3)调用结束并返回 6。

（8）程序回到 main 函数，执行 cout 语句，输出结果 n!=6，程序结束。递归过程如图 6-2 所示。

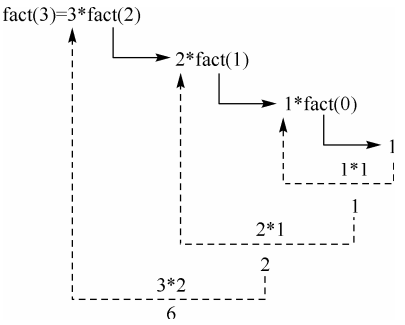


图 6-2　递归过程**错误!**

【例 6.14】 计算 $f(x)=x_n$，要求采用递归调用函数完成。

分析：求 $x_n=x \cdot x_{m-1}$，只需要知道 x_{m-1}，设函数 f(x, n)，求取 x_n，此时 n=m。

求 $x_{m-1}= x \cdot x_{m-2}$，只需要知道 x_{m-2}，自身调用 f(x, n)，求取 x_{m-1}，此时 n=m–1。

依次类推：$x_{m-2}=x \cdot x_{m-3}$，只需要知道 x_{m-3}……

x_0 = 1, 0 为递归调用的终止条件。

程序如下：

```
include<iostream>
using namespace std;
long f(int x,int n)
{int y;
if(n>0)y=f(x,n-1)*x;
else y=1;
return y;
}
void main( )
{int x,n;
cout<<"输入 x 和 n 的值"<<endl;
cin>>x>>n;cout<<"x 的 n 次方为:"<<f(x,n)<<endl;
}
```

6.5 重载函数

函数重载是 C++支持的一种特殊函数，又称为函数的多态性，是指在相同的声明域中有同名的函数，但函数类型或参数表必须不同。例如，下面是合法的重载函数：

```
int max(int,int);
double max(double,double);
float max(float,float,float);
```

重载函数的类型，即函数的返回类型可以相同，也可以不同。但如果仅仅是返回类型不同而函数名相同、形参表也相同，则是不合法的，编译器会报“语法错误”。例如：

```
int max(int a, int b);
double max(int a, int b);
```

在调用一个重载函数 max()时，编译器必须判断函数名 max 到底是指哪个函数。它是通过编译器，根据实参的个数和类型对所有 max()函数的形参一一进行比较，从而调用一个最匹配的函数。

【例 6.15】 函数重载举例。

```
#include <iostream>
using namespace std;
float add(float x, float y)
{   return (x+y); }
int add(int x, int y)           //形参类型不同
{   return (x+y); }
int add(int x, int y, int z)    //形参个数不同
{   return (x+y+z); }
void main( )
{   float a,b,c;
    cout<<"输入两个浮点数:"<<endl;
    cin>>a>>b;
    c=add(a,b);
    cout<<"这两个浮点数的和是:"<<c<<endl;
```

```
    int m,n,s;
    cout<<"输入两个整数:"<<endl;
    cin>>m>>n;
    s=add(m,n);
    cout<<"这两个整数的和是:"<<s<<endl;
    int i,j,k,l;
    cout<<"输入三个整数:"<<endl;
    cin>>i>>j>>k;
    l=add(i,j,k);
    cout<<"这三个整数的和是:"<<l<<endl;
}
```

6.6　模板函数

1．模板的概念

前面讲到的重载函数，可以让程序员对具有相同功能的函数进行重载。例如：

```
int max(int x,int y)
{return(x>y)?x:y;}
float max(float x,float y)
{return (x>y)?x:y;}
double max(double x,double y)
{return (x>y)?x:y;}
```

这些函数的功能相同，只是参数的类型和返回值类型不同，能否为这些函数只写一套代码？答案就是使用模板。即将数据类型改为一个设计参数，这种类型的程序设计也称为参数化(parameterize)程序设计。例如：

```
template<class type>
type max(type a,type b)
{return(a>b)?a,b;}
```

所谓模板，就是写一个函数模子，用这个模子可以完成许多功能相同，即参数类型和返回值不同的函数。所以模板真正解决了代码的可重用性问题。

函数重载和函数模板有着本质的区别。函数重载是指用同一个名字定义不同的函数。而函数模板是指用一个名字定义不同的函数，这些函数功能相同，但参数类型和返回值类型不同。

模板分函数模板和类模板，C++允许用户分别用它们构造出模板函数和模板类。当模板被实例化时，实际的内置或用户定义类型将替换模板的参数类型。如 int、double、char 等都是有效的模板实参类型。

2．函数模板的定义

```
template<类型 1　变量 1, 类型 2　变量 2, …… ><返回值类型> <函数名>(函数形参表)
{　　函数体；}
```

其中，template 是声明模板的关键字，表示声明一个模板。

模板中的类型参数由一个普通的参数声明构成。模板类型参数表示该参数名代表一个潜在的值，而该值代表模板定义中的一个常量，如 size 就是一个模板类型参数，它代表 x 指向的数组长度：

```
template <class Type,int size>
Type max(const Type(&x)[size])//定义一个 Type 类型的函数，寻找数组中的最大值
{Type max=x[0];
for(int i=1;i<size;i++)
 if(x[i]>max)max=x[i];
return max;
}
```

在这个例子中，Type 表示 max()的返回类型，size 表示 x 引用数组的长度。在程序运行过程中，Type 会被各种内置类型和用户定义的类型所替代，而 size 会被常量值所取代，这些常量值是由实际的 max()决定的（注：函数的两种用法是调用它和取它的地址），类型和值的替换过程称为模板的实例化。

当函数模板 max()被实例化时，size 的值会被编译成已知的常量，函数定义或声明跟在模板参数表后，除模板参数是类型标识符或常量外，函数模板的定义与一般函数的定义相同。

【例 6.16】 编写求数组中最大值的函数，设计成函数模板。

```
#include <iostream>
using namespace std;
template <class Type,int size>
Type max(const Type(&x)[size])  //寻找数组中的最大值
{Type max=x[0];
for(int i=1;i<size;i++)
 if(x[i]>max)max=x[i];
return max;
}
void main()
{ int a[]={10,6,8,14,5,25};
double d[]={10.5,23.4,56.7,44.5,11.6,23.9};
int i=max(a);                    //用数组实例化 max()
cout<<"整数数组 a 的最大值是:"<<i<<endl;
double df=max(d);                //用数组 d 实例化 max()
cout<<"实型数组 d 的最大值是:"<<df<<endl;
}
```

此程序的运行结果为：

整数数组 a 的最大值是：25

实型数组 d 的最大值是：56.7

在运行该程序中的 int i=max(a);语句时，被实例化的 max()的整型实例，这里 Type 被 int、size 被 6 取代。

3．使用函数模板

函数模板只是说明，不能直接执行，需要实例化为模板函数后才能执行。当一个名字被声明为模板参数后，它就可以被使用了，一直到模板声明或定义结束。模板类型参数被用做类型标识符，可以出现在模板定义的余下部分。其使用方式与内置或用户定义的类型完全一样，如用来声明变量和强制类型转换。模板的非类型参数被用做一个常量，可以出现在模板定义的余下部分，它用在要求常量的地方，或在声明数组中指定数组的大小，或作为枚举常量的初始值。

在函数模板定义中声明的对象或类型不能与模板参数同名：

```
template <class Type>
Type max(Type a,Type b)                    //错误的声明
{  //重新声明模板参数 Type
   Typedef double Type;
   Type temp=a>b?a:b;
   return  temp
}
```

模板类型参数名可以用来指定函数模板的返回类型。

```
template <class T1, class T2, class T3>     //T2、T3 表示参数类型
       T1 max(T2,T3);                       //T1 表示 max( )函数的类型
```

模板参数名在同一模板参数表中只能被使用一次。下面的代码是错误的:

```
template <class T1, class T1>
T1 max(T1 ,T1);
```

但模板参数名可以在多个函数模板声明或定义之间被重复使用。例如：

```
template <class T1>
T1 max(T1,T1);
template <class T1>
T1 add(T1,T1);
```

如果一个函数模板有一个以上的模板类型参数，则每个模板类型参数前都必须有关键词 class 或 typename。例如：

```
template <typename T, class X>          //typename 和 class 可以混用
T sum(T*,X);                            //错误的说明
```

可以改为：

```
<typename T, class X>或<typename T, typename X>
template  <typename T, X>
T sum(T*, X);
```

在函数模板参数表中，关键词 typename 和 class 意义相同，可以互换使用。关键词 typename 是在 VS 中新加入到标准 C++中的。

4．函数模板应用举例

函数模板的应用，使得程序员能够用不同类型的参数调用相同的函数，由编译器决定该用哪种类型，并从模板函数中生成相应的代码。

【例 6.17】 定义一个函数后，把这个函数改为模板函数。

```
#include <iostream>
using namespace std;
#include<string>
void printstr(const string& str)
{ cout<<str<<endl;}
```

```
void main( )
{ string str("I am a student");
printstr(str);
}
```

将这个函数改为模板：

```
#include <iostream>
using namespace std;
#include<string>
template<typename T>
void printstr(const T& str)
{ cout<<str<<endl;}
void main( )
{ string str("I am a student");
printstr(str);
}
```

【例 6.18】 求三个数或字符串中的最大值。

```
#include <iostream>
using namespace std;
#include<string>
template<typename T>
T max(T a,T b,T c)
{ if(b>a)a=b;
if(c>a)a=c;
return a;
}
void main( )
{int x,y,z,k;
cin>>x>>y>>z;                //输入 3 个整数
k=max(x,y,z);
cout<<"最大整数值:"<<k<<endl;
string st1,st2,st3,str;
cin>>st1>>st2>>st3;          //输入 3 个字符串
str=max(st1,st2,st3);
cout<<"最大字符串是:"<<str<<endl;
}
```

6.7 内联函数

在 C++中，为了解决一些频繁调用的小函数大量消耗栈空间或称为栈内存的问题，特别引入了 inline 修饰符，表示为内联函数。所谓栈空间是指存放程序的内部数据的内存空间。在计算机系统中，栈空间是有限的，如果频繁大量地使用就会造成因栈空间不足，导致程序出错，函数的死循环、递归调用的最终结果都会导致栈内存空间枯竭。下面来看一个例子：

【例 6.19】 判断数组元素的奇偶性。

```
#include <iostream>
#include <string>
using namespace std;
  inline string pac(int x); //定义一个 string 类型的内联函数
    int a[10]={3,67,45,22,47,32,90,14,13,47};
void main( )
{  for (int i=0;i<10;i++)
     cout << a[i] << ":" << pac(a[i]) << endl;
    }
string pac(int x)            //这里不用再次 inline，也可以加上 inline
{     return (x%2>0)?"奇":"偶";    }
```

上面的例子就是标准的内联函数的用法，使用 inline 修饰带来的好处是在每个 for 循环的内部所有调用 pac()的地方都换成了(x%2>0)?"奇":"偶"，这样就避免了频繁调用函数对栈内存重复开辟所带来的消耗。

需要注意的是引入内联函数的目的是为了解决程序中函数调用的效率问题。内联函数只适合函数体内代码简单的函数使用，不能包含复杂的结构控制语句如 for、while、goto、switch 等，并且内联函数本身不能是直接递归函数。在程序中，调用其函数时，该函数在编译时被替代，而不是像一般函数那样是在运行时被调用。

内联函数可以在一开始仅定义或声明一次，但必须在函数被调用之前定义或声明。否则，编译器不认为那是内联函数，仍然如同对普通函数那样处理该函数的调用过程。从用户的角度看，调用内联函数和一般函数没有任何区别。

6.8　函数指针

在程序运行中，函数代码是程序的算法指令部分，它们和数组一样也占用存储空间，并都有相应的地址。可以使用指针变量指向数组的首地址，也可以使用指针变量指向函数代码的首地址，指向函数代码首地址的指针变量称为函数指针。因而“函数指针”就是一个指针变量，该指针变量指向函数，而函数的入口地址就是函数指针所指向的地址。有了指向函数的指针变量后，可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样。

函数指针有两个用途：调用函数和做函数的参数。

1．函数指针声明

函数指针的声明方法为:

```
<函数类型>(*指针变量名) (<形参列表>);
```

其中：

函数类型：说明函数的返回类型；

*指针变量名：表示指向函数指针的名字；

形参列表：表示指针变量指向的函数所带的参数列表。

例如：

```
int fun(int x);                    //声明一个函数
int (*f) (int x);                  //声明一个函数指针
f=fun;                             //将 fun 函数的首地址赋给指针 f
```

赋值时函数 fun 不带括号，也不带参数，由于 fun 代表函数的首地址，因此经过赋值以后，指针 f 就指向函数 fun(int x)代码的首地址。

注意：如果将函数指针的定义写成 int *f(int x);是错误的，因为按照结合性和优先级来看是先和()结合，然后变成了一个返回整形指针的函数，而不是函数指针，这一点尤其需要注意。

另外，函数括号中的形参不能省略；在定义函数指针时，函数指针和它指向的函数的参数个数和类型都应该是一致的；函数指针的类型和函数的返回值类型也必须是一致的。

2．函数指针调用函数方法

【例 6.20】 任意输入 n 个数，找出其中最大数，并输出最大数值。

```
#include <iostream>
using namespace std;
int f(int x,int y);
int a,b,z;
void main( )
{     int i;
       int (*p)(int,int);    /*定义函数指针 */
       cin>>a;
       p=f;                  /*给函数指针 p 赋值，使它指向函数 f */
       for(i=1;i<9;i++)
       {    cin>>b;
          a=(*p)(a,b);       /*通过指针 p 调用函数 f */
       }
       cout<<"The Max Number is:"<<a;
}
int f(int x,int y)
{  z=(x>y)?x:y;
   return z;}
```

运行结果为：

23 –40 99 45 1 –47 43 234↙

The Max Number is:234

这里 p 是指向函数的指针变量，所以可把函数 f(int x, int y)赋给指针 p 作为入口地址，以后就可以用 p 来调用该函数。实际上 p 和 f 都指向同一个入口地址，不同的是 p 为一个指针变量，可以指向任何函数。在程序中把哪个函数的地址赋给它，它就指向哪个函数，而后用指针变量调用它，不过注意，指向函数的指针变量没有++和—运算，用时要小心。

注意：定义指针函数时，指针函数中的形参不能省略，这是 VC++ 2005 与 VC 6.0 的区别。

【例 6.21】 观察下列程序的运行结果。

```
#include <iostream>
using namespace std;
```

```
int test(int a);
void main( )
{
cout<<test<<endl;       //显示函数地址
   int (*fp)(int a);
   fp=test;             //将函数 test 的地址赋给函数指针 fp
   cout<<fp(15)<<"--"<<(*fp)(20)<<endl;
}
int test(int a)
{
   return a;
}
```

程序运行结果：

```
004110FF
15–20
```

3．函数指针类型

可以利用 typedef 简化函数指针的定义，这种方式在定义多个函数指针时会方便一些。函数指针的一般格式为:

```
typedef int (*fun_ptr)(int,int);
```

然后声明变量并赋值：

```
fun_ptr max_func=max;
```

也就是说，赋给函数指针的函数应该和函数指针所指的函数原型是一致的。

【例 6.22】 改写例 6.21 的程序。

```
#include <iostream>
using namespace std;
  int test(int a);
    void main( )
{
cout<<test<<endl;
typedef int (*fp)(int a);//定义一个函数指针类型，这个类型是自己定义的，类型名为 tp
   fp tp;               //利用自定义的类型名 tp 定义一个 fp 的函数指针
   tp=test;
  cout<<tp(15)<<"--"<<(*tp)(20)<<endl;
}
int test(int a)
{
   return a;
}
```

程序运行结果：

```
004110FF
15–20
```

4．指针函数

指针函数和函数指针的区别：首先这两个概念都是简称，指针函数是指带指针的函数，即本质是一个函数。因为函数一般都有返回类型(如果不返回值，则为无类型)，所以指针函数返回类型是某一类型的指针。

其定义格式为：

```
返回类型标识符*返回名称(形式参数表)
  {函数体}
```

返回类型可以是任何基本类型和复合类型。返回指针的函数用途十分广泛。事实上，每个函数，即使它不带有返回某种类型的指针，它本身都有一个入口地址，该地址相当于一个指针。比如函数返回一个整型值，实际上也相当于返回一个指针变量的值，不过这时的变量是函数本身而已，而整个函数相当于一个“变量”。

【例 6.23】 返回指针函数的例子。

```
#include <iostream>
using namespace std;
#include<string.h>
float *find(float (*point)[4],int n);
static float score[][4]={{70,75,83,94},{47,81,33,76},{56,76,58,75}};
void main()
{  float *p;
int i,m;
cout<<"Enter the number to be found:";
cin>>m;
if (m>sizeof(score)/(sizeof(score[0]))-1)    //判断数组是否越界
{   cout<<"m is overflow the row of array!";
    return;
}
p=find(score,m);
for(i=0;i<4;i++)
cout<<" "<<*(p+i);
}
float *find(float(*point)[4],int n)          /*定义指针函数*/
{   float *pt;
    pt=*(point+n);
    return(pt);
}
```

本例中，输入的数据只能是 0、1、2。

学生学号从 0 号算起，函数 find()被定义为指针函数，形参 point 是指针指向包含 4 个元素的一维数组的指针变量。p+1 指向 score 的第一行。*(p+1)指向第一行的第 0 个元素。pt 是一个指针变量，它指向浮点型变量。main()函数中调用 find()函数，将 score 数组的首地址传给 p。

本 章 小 结

1. C++程序是由函数构成的。

2. C++必须知道函数的返回类型，以及形参的个数、类型和次序。函数必须先定义后调用。如果函数的定义出现在函数调用之后，则必须在程序的开始部分用函数原型进行说明。

3. 局部变量是在函数内定义的，只能在定义该变量的函数内访问。

4. 全局变量是在所有函数外定义的，其作用域和生命周期均为全局。

5. 静态内部变量在函数内部定义，其作用域为函数内部。在函数内部定义的静态变量的初始值只在第一次被调用时有效，其后的调用随静态变量值的改变而改变。

6. 静态全局变量与全局变量类似，具有全局作用域。其差别是：静态全局变量的作用域为定义该静态变量的源程序文件；而全局变量的作用域为程序的整个源程序文件。

7. 函数可递归调用，但不能嵌套定义。

8. 内联函数是为了提高编程效率而设置的。它克服了#define 宏定义可能带来的副作用。

9. 函数重载允许用一个函数名定义多个函数。连接程序会根据传递的实参数目、类型和顺序调用相应的函数。函数重载使程序设计简单化，程序员只需要记住一个函数名就可完成一系列的相关任务。

10. 函数中的形参可以被定义为带有默认值的形参，当调用函数没有给该形参赋值时，就使用该默认值。

11. 作用域规定程序中变量和函数的有效范围。它给可见性提供了依据，使得在具有多文件的大型软件开发中，增加了使用数据的灵活性和安全性。

习　题　6

一、选择题

1. 以下关于函数的说法，正确的是(　　)。
   A. 如果形参与实参类型不一致，以实参类型为准
   B. 如果函数值的类型与返回值类型不一致，以函数值类型为准
   C. 形参的类型说明可以放在函数体内，以实参类型为准
   D. return 后面的值不能为表达式
2. 下列叙述中，不正确的是(　　)。
   A. 一个函数可以有多个 return 语句　　B. 函数可以通过 return 语句返回数据
   C. 必须有一个独立的语句来调用函数　　D. 函数 main 可以带有参数
3. 下列关于变量的叙述中不正确的是(　　)。
   A. 自动变量和外部变量的作用域为整个程序文件
   B. 函数内定义的静态变量的作用域为整个程序文件
   C. C++语言中将变量分为 auto、static、eatern、register 四种存储类型
   D. 外部静态变量的作用域为定义它的文件内

4. 下面关于函数重载的说法正确的是(　　)。
   A. 重载函数必须有不同的形参列表
   B. 重载函数必须有不同类型的返回值类型
   C. 重载函数的形参个数必须不同
   D. 重载函数名可以不同
5. 下列有关设置函数默认值的描述中，正确的是(　　)。
   A. 对设置函数参数默认值的顺序没有任何规定
   B. 函数具有一个参数时不能设置默认值
   C. 默认参数要设置在函数的定义语句中，而不能设置在函数的说明语句中
   D. 设置默认参数可以使用表达式，但表达式中不可用局部变量
6. 在函数的引用调用中，实参和形参正确的用法是(　　)。
   A. 常量值和变量　　B. 地址值和常量值
   C. 变量值和引用　　D. 地址值和引用

二、判断题

1. 内联函数是为了提高编程效率而实现的，它克服了用#define 宏定义所带来的弊病。
2. C++函数必须有返回值，否则不能使用函数。
3. 函数的定义可以嵌套，函数的调用不可以嵌套。
4. 函数重载是指两个或两个以上的函数取相同的函数名，但形参的个数或类型不同。
5. 参数的默认值只能在定义函数时设置。
6. 在不同的函数中，可以使用相同名字的变量。
7. 函数的传址调用和引用调用都可以在被调用函数中改变调用函数的参数值。

三、写出程序的运行结果或功能

(1)

```
#include<iostream>
using namespace std;
int a=5;
void main()
{ int i,a=10,b=20;
cout<<a<<","<<b<<endl;
  { int a=0,b=0;
    for(i=1;i<9;i++)
      {a+=i;
       b+=a;
      }
  }
  cout<<a<<","<<b<<","<<::a<<endl;
}
```

(2)

```
long gcd1(int a,int b)
{if (a%b==0)
   return b;
```

```
else return gcd1(b,a%b);
}
```

(3)

```
long gcd2(int a,int b)
{ int temp;
while(b!=0)
{ temp=a%b;
  a=b;
  b=temp;
}
return a;
}
```

(4)

```
#include<iostream>
using namespace std;
int fun(int x)
{ cout<<x<<' ';
if(x<=0){cout<<endl;return 0;}
else return x*fun(x-1);
}
void main( )
{int x=fun(5);
cout<<x<<endl;
}
```

(5)

```
void contrary(int x)
{ if(x){cout<<x%10;
    contrary( x/10);}
else cout<<endl;
}
```

四、编写程序

1. 编写一个判断素数的函数，证明哥德巴赫猜想：任何一个充分大的偶数(大于 6)，都可以分解成两个素数的和。如 int prime(int n)，当参数值为素数时，返回 1，否则返回 0。

2. 编写一个函数，由实参传递一个字符串，统计该字符串中字母、数字的个数，并在主函数中输出结果。

3. 编写一个函数，从一个二维整型数组中查找具有最大值的元素及其位置。

4. 使用递归调用的方法将一个 n 位整数转换为字符串。

5. 使用函数重载的方法定义两个函数，分别计算两个整数间的差和两个实型数之间的差。

6. 从键盘上输入 10 个整数，去掉重复的，将剩余的数由大到小排序。

第 7 章　结构体与联合

计算机程序设计处理的核心内容是数据，而数据具有不同的形式，比如企业员工的姓名为字符串，年龄为整型，为了描述不同类型的数据，C++语言提供了整型、字符型、浮点型、布尔型等数据类型，这些称为基本数据类型。但是有些程序处理的内容较为复杂，不能单独使用一种基本数据类型定义，比如企业管理程序中处理的主要内容为“员工”，一个“员工”拥有员工号、姓名、性别、年龄、工资等不同属性，对“员工”就不能只使用一种基本数据类型来定义。为了描述这些较为复杂的内容，C++语言允许用户自己定义某些数据类型，这些数据类型称为用户自定义类型(user-defined type，UDT)。C++语言中用户自定义数据类型包括数组(array)、结构体(structure)、联合(union)、枚举(enum)、类(class)等。本章介绍结构体、联合和枚举类型。

7.1　结构体类型

7.1.1　结构体的定义

结构体与数组类型相似，都是用来存储数据的多个元素。两者的主要区别为：数组中的所有元素必须属于同一种类型；而结构体中的元素可以是不同类型的，所以结构体一般适用于存储具有多种属性的记录。结构体把相关的不同类型数据“组合”在一起，作为一个整体来使用。比如企业管理程序中“员工”，其员工号、姓名可以使用字符串描述，年龄可以用整型描述，工资可以用实型描述，图 7-1 表示了一个拥有 4 个属性的员工结构体。

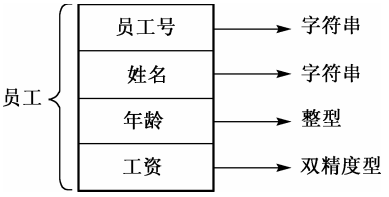


图 7-1　“员工”中的不同属性

结构体中的不同属性称为成员。定义一个结构体类型的格式为：

```
struct 结构体类型名
  {
    数据类型名 1  成员表列 1;
     数据类型名 2  成员表列 2;
    …
    数据类型名 n  成员表列 n;
};
```

图 7-1 中的“员工”结构体类型可以定义如下：

```
struct Employee              //结构体类型名为 Employee
{
    string employeeid;       //Employee 结构成员 1 员工号为字符串
    string name;             //Employee 结构成员 2 姓名为字符串
```

```
    int age;                    //Employee 结构成员 3 年龄为整型
    double wage;                //Employee 结构成员 4 工资为双精度型
};
```

以上过程表示在程序中定义了一个结构体类型，其类型名为 Employee，该结构体类型包括 4 个成员，分别为 employeeid、name、age 和 wage。需要注意的是，这个过程只是向 C++编译系统定义了一个新的类型，还不能立刻使用该结构体的成员。此时 Employee 只是一个类型名，相当于 C++编译系统提供的 int、double、char 等标准类型。只有使用 Employee 类型声明了一个结构变量后才能使用其中的成员。

关于结构体的定义，需要注意以下几点:

(1) struct 为定义结构体的关键字，不能省略。struct 后边的“结构体类型名”是所定义的结构体类型的名字。

(2) 结构体的成员需要用一对大括号括起来，在最后一个大括号末尾需要分号结尾。

(3) 结构体中不同成员的定义一般写在不同行，每一行由分号结尾；不同成员的定义也可以写在同一行，中间由分号隔开。

(4) 结构体中的成员可以为标准的数据类型，也可以是另一个结构体类型的变量。例如:

```
struct Date          //定义一个结构体类型 Date
{
  int month;
  int day;
  int year;
};
struct Employee      //定义一个结构体类型 Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
   Date worktime;    //使用 Date 类型声明一个变量 worktime，表示“参加工作时间”
};
```

7.1.2　结构体变量的定义和初始化

定义了结构体类型后就可以使用该类型进行变量的声明，只有使用结构体类型进行变量声明后，C++编译系统才会为该结构类型变量分配内存单元。结构体类型变量的声明方法主要有以下几种：

(1) 先定义结构体类型再声明结构体变量

如果使用前边定义的 Employee 结构体类型进行变量声明，语句为：

```
Employee  employee1, employee2;
```

表示使用 Employee 结构体类型声明了两个变量 employee1 和 employee2。这种声明方式与 C++标准类型变量声明是一样的。

C 语言中结构体变量声明时需要在结构体类型前面加上关键字 struct，C++中也允许这种方法，例如：

```
struct  Employee  employee1, employee2;
```

但是 C++语言提倡使用不加关键字的方法，这种方法使得结构体变量的声明与标准类型变量的声明一致，更加容易理解。

(2) 定义结构体类型的同时声明变量

```
struct Employee
{  string employeeid;
   string name;
   int age;
   double wage;
}  employee1, employee2;
```

(3) 直接声明结构体类型变量

```
struct
{
   string employeeid;
   string name;
   int age;
   double wage;
}  employee1, employee2;
```

这种方法直接声明了结构体类型变量，而没有给出结构体类型名，此时在当前程序中其他地方不能再使用该结构体声明变量。C++语言中提倡先定义数据类型再声明变量，这样程序结构清楚，所以不提倡使用直接声明的方法声明结构体类型变量。

声明结构体类型变量时可以同时指定其成员的初始值，称为结构体变量的初始化。如果使用第(1)种结构体变量声明方式，其初始化过程如下：

```
Employee  employee1={"20100001", "王军", 25, 2100.5};
```

如果使用第(2)种结构体声明方式，其初始化过程如下：

```
struct Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
}  employee1={"20100001", "王军", 25, 2100.5};
```

7.1.3 结构体变量的引用

声明一个结构体变量后就可以使用该变量。引用结构体变量一般指引用该结构体变量的成员。结构体变量成员的引用格式为：

```
结构体变量名.成员名
```

其中“.”为成员访问运算符(member access operator)，在 C++的所有运算符中与圆括号运算符“()”、下标运算符“[ ]”同属最高优先级别。

如给结构体变量 employee1 的成员 employeeid 赋值可以写成以下方式：

```
employee1. employeeid = "20100001";
```

如果某个结构体变量中的一个成员本身也是结构体类型，则其成员的引用方法是逐级使

用成员访问运算符，直到找到最低级别的成员。如结构体变量 employee1 中包括了成员 worktime，而 worktime 本身为一个 Date 结构体类型变量，如果引用 employee1 的 worktime 成员中的 year 成员，其访问方式为：

```
employee1. worktime.year = 2010;
```

可以使用一个结构体变量给另一个结构体变量赋值。需要注意的是，相互赋值的两个结构体变量必须为同一结构体类型。

假设使用 Employee 结构体类型声明了两个变量 employee1 和 employee2，并分别进行了成员赋值操作。可以使用以下语句把 employee1 的每个成员的值赋给 employee2 中相应的成员：

```
employee2 = employee1;
```

以上代码相当于：

```
employee2.employeeid = employee1.employeeid;
employee2.name = employee1.name;
employee2.age = employee1.age;
employee2.wage = employee1.wage;
```

【例 7.1】 使用结构体保存员工的相关信息。

```
#include <iostream>
#include <string>
using namespace std;
struct Date         //定义结构体类型 Date
{
  int month;
  int day;
  int year;
};
struct Employee     //定义结构体类型 Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
   Date worktime;
} employee1={"20100001", "王军", 25, 2100.5, 2, 20, 2010};
int main()
{
    Employee employee2;
    employee2.employeeid = "20100002";
    employee2.name = "李明";
    employee2.age = 28;
    employee2.wage = 3500.5;
    employee2.worktime.month = 3;
    employee2.worktime.day = 15;
    employee2.worktime.year = 2010;
    cout<<"员工"<<employee2.name<<"的基本信息为:"<<endl;
    cout<<"员工号:"<<employee2.employeeid<<endl;
```

```
        cout<<"姓名:"<<employee2.name<<endl;
        cout<<"年龄:"<<employee2.age<<endl;
        cout<<"工资:"<<employee2.wage<<endl;
        cout<<"参加工作时间:"<<employee2.worktime.year<<"-"
        <<employee2.worktime.month<<"-"
        <<employee2.worktime.day<<endl;
        employee2 = employee1;
        cout<<"员工"<<employee2.name<<"的基本信息为:"<<endl;
        cout<<"员工号:"<<employee2.employeeid<<endl;
        cout<<"姓名:"<<employee2.name<<endl;
        cout<<"年龄:"<<employee2.age<<endl;
        cout<<"工资:"<<employee2.wage<<endl;
        cout<<"参加工作时间:"<<employee2.worktime.year<<"-"
        <<employee2.worktime.month<<"-"
        <<employee2.worktime.day<<endl;
        return 0;
    }
```

将输出如下的结果：

员工李明的基本信息为：
员工号：20100002
姓名：李明
年龄：28
工资：3500.5
参加工作时间：2010-3-15
员工王军的基本信息为：
员工号：20100001
姓名：王军
年龄：25
工资：2100.5
参加工作时间：2010-2-20

7.1.4 结构体数组

C++语言中用数组来表示具有一组相同类型的数据集合。如果数组中的元素为结构体类型，那么该数组就称为结构体数组，其中的每个元素都是一个结构体变量。

结构体数组的定义方式与普通数组类似，假设程序中已经定义了某一结构体类型，那么定义该结构体数组的格式为：

```
结构体类型名 数组名[常量表达式]
```

其中数组名和常量表达式与普通数组的定义方式相同，结构体类型名为程序中事先定义好的结构体类型。如使用 7.1.1 中定义的 Employee 结构体类型可以定义以下数组。

```
Employee employee[10];
```

通过该语句定义了一个结构体数组 employee，其中包含有 10 个元素，元素的下标从 0 到 9，每个元素都是一个 Employee 结构体类型的变量。

可以在定义结构体数组时给每个结构体变量进行初始化，也可以先定义结构体数组，再分别给每个结构体变量赋值。需要注意的是，此时给结构体变量赋值需要分别给结构体变量的各个成员赋值。例如：

```
struct Employee    //定义结构体类型 Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
 } employee[3] ={
     {"20100001","王军", 25, 2010.5},
     {"20100002","李明", 24, 2310.5},
     {"20100003","刘云", 26, 2910.5}
    };
```

如果先定义了结构体数组，再给数组元素赋值，需要如下进行：

```
Employee employee[3];
//给第一个数组元素赋值
employee[0].employeeid ="20100001";
employee[0].name ="王军";
employee[0].age =25;
employee[0].wage =2010.5;
```

而不能写成：

```
employee[0] = {"20100001","王军", 25, 2010.5};
```

【例 7.2】 假设某企业 2010 年有 3 名新员工，设计程序使用结构体保存新员工的相关信息，计算所有新员工的平均工资。

```
#include <iostream>
#include <string>
using namespace std;
struct Employee     //定义结构体类型 Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
 };
int main( )
{
    int i;
    float sumwage,avewage;
    Employee employee[3] ={
     {"20100001","王军", 25, 2010.5},
     {"20100002","李明", 24, 2310.5},
     {"20100003","刘云", 26, 2910.5}
```

```
    };
    sumwage = 0;
    for(i=0;i<3;i++)
    {
        sumwage += employee[i].wage;
    }
    avewage = sumwage/3;
    cout<<"员工"<<employee[0].name<<"、"
    <<employee[1].name<<"、"
    <<employee[2].name<<"的平均工资为:"<<endl;
    cout<<avewage<<endl;
    return 0;
}
```

将输出如下的结果：

员工王军、李明、刘云的平均工资为：

2410.5

7.1.5 结构体与函数

结构体类型只要定义以后就可以和 C++语言的标准数据类型一样使用了，所以结构体也可以作为函数的参数来使用。

【例 7.3】 将例 7.1 中输出结构体成员部分改成函数来处理。

```
#include <iostream>
#include <string>
using namespace std;
struct Date         //定义结构体类型 Date
{
  int month;
  int day;
  int year;
};
struct Employee     //定义结构体类型 Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
   Date worktime;
} employee1={"20100001", "王军", 25, 2100.5, 2, 20, 2010};

void printemployee(Employee employee)  //参数为 Employee 结构体类型变量
{
    cout<<"员工"<<employee.name<<"的基本信息为:"<<endl;
    cout<<"员工号:"<<employee.employeeid<<endl;
    cout<<"姓名:"<<employee.name<<endl;
    cout<<"年龄:"<<employee.age<<endl;
    cout<<"工资:"<<employee.wage<<endl;
```

```
    cout<<"参加工作时间:"<<employee.worktime.year<<"-"
    <<employee.worktime.month<<"-"
    <<employee.worktime.day<<endl;
}
int main()
{
    Employee employee2;
    employee2.employeeid = "20100002";
    employee2.name = "李明";
    employee2.age = 28;
    employee2.wage = 3500.5;
    employee2.worktime.month = 3;
    employee2.worktime.day = 15;
    employee2.worktime.year = 2010;
    printemployee(employee2);
    employee2 = employee1;
    printemployee(employee2);
    return 0;
}
```

将输出如下的结果：

员工李明的基本信息为：
员工号：20100002
姓名：李明
年龄：28
工资：3500.5
参加工作时间：2010-3-15
员工王军的基本信息为：
员工号：20100001
姓名：王军
年龄：25
工资：2100.5
参加工作时间：2010-2-20

在例 7.3 中输出函数 printemployee 的参数为 Employee 结构体类型变量，在函数中可以直接访问该变量的各个成员。在 C++语言中结构体作为函数的参数主要有三种方式：

(1) 将结构体变量作为函数参数；
(2) 将指向结构体变量的指针作为函数参数；
(3) 将结构体变量的引用变量作为函数参数。

后两种方式在介绍结构体指针时再做具体介绍。

结构体变量可以作为函数参数，也可以作为函数返回值使用。如果函数返回结构体类型变量，该结构体类型必须先定义。

【例7.4】 在例 7.3 的基础上增加一个修改员工工资的函数，函数接收一个 Employee 结构体变量和需要修改的工资数为参数，修改工资后返回新的 Employee 结构体变量。

```
#include <iostream>
#include <string>
using namespace std;
struct Date         //定义结构体类型 Date
{
  int month;
  int day;
  int year;
};
struct Employee     //定义结构体类型 Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
   Date worktime;
} employee1={"20100001", "王军", 25, 2100.5, 2, 20, 2010};
void printemployee(Employee employee)  //参数为 Employee 结构类型变量
{
    cout<<"员工"<<employee.name<<"的基本信息为:"<<endl;
    cout<<"员工号:"<<employee.employeeid<<endl;
    cout<<"姓名:"<<employee.name<<endl;
    cout<<"年龄:"<<employee.age<<endl;
    cout<<"工资:"<<employee.wage<<endl;
    cout<<"参加工作时间:"<<employee.worktime.year<<"-"
    <<employee.worktime.month<<"-"
    <<employee.worktime.day<<endl;
}
//函数返回值为 Employee 结构类型变量
Employee modifywage(Employee employee, float addedvalue)
{
    employee.wage += addedvalue;
    return employee;
}
int main()
{
    printemployee(employee1);
    employee1 = modifywage(employee1,100);
    printemployee(employee1);
    return 0;
}
```

将输出如下的结果：

员工王军的基本信息为：
员工号：20100001
姓名：王军
年龄：25

工资：2100.5
参加工作时间：2010-2-20
员工王军的基本信息为：
员工号：20100001
姓名：王军
年龄：25
工资：2200.5
参加工作时间：2010-2-20

7.1.6　结构体指针

C++中基本数据类型变量的指针用来表示该变量所占据的内存的起始地址，同样一个结构体类型变量的指针表示该结构体变量所在内存的起始地址。指向结构体变量的指针称为结构体指针变量。当声明了结构体变量后，就可以使用该指针变量来访问它所指向的结构体变量中的成员。

声明结构体指针变量的格式为：

```
结构体类型名　*结构体指针变量名;
```

【例 7.5】 使用结构体变量指针。

```
#include <iostream>
#include <string>
using namespace std;
struct Employee    //定义结构体类型 Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
 } employee1={"20100001", "王军", 25, 2100.5};
int main( )
{
    Employee *p;
    p = &employee1;
    cout<<"员工"<<employee1.name<<"的基本信息为:"<<endl;
    cout<<"员工号:"<<employee1.employeeid<<endl;
    cout<<"姓名:"<<employee1.name<<endl;
    cout<<"年龄:"<<employee1.age<<endl;
    cout<<"工资:"<<employee1.wage<<endl;
    cout<<"员工"<<(*p).name<<"的基本信息为:"<<endl;
    cout<<"员工号:"<<(*p).employeeid<<endl;
    cout<<"姓名:"<<(*p).name<<endl;
    cout<<"年龄:"<<(*p).age<<endl;
    cout<<"工资:"<<(*p).wage<<endl;
    cout<<"员工"<<p->name<<"的基本信息为:"<<endl;
```

```
    cout<<"员工号:"<< p->employeeid<<endl;
    cout<<"姓名:"<< p->name<<endl;
    cout<<"年龄:"<< p->age<<endl;
    cout<<"工资:"<< p->wage<<endl;
    return 0;
}
```

将输出如下的结果：

```
员工王军的基本信息为：
员工号：20100001
姓名：王军
年龄：25
工资：2100.5
员工王军的基本信息为：
员工号：20100001
姓名：王军
年龄：25
工资：2100.5
员工王军的基本信息为：
员工号：20100001
姓名：王军
年龄：25
工资：2100.5
```

上述程序中，首先定义了 Employee 结构体类型，声明 employee1 变量的同时进行了初始化，在主函数中声明结构体变量指针 p，然后给 p 赋值为结构体变量 employee1 的起始地址。也可以在声明指针 p 的同时赋值，语句如下：

```
Employee *p = &employee1;
```

通过程序可以看出，直接使用结构体变量进行成员输出和使用结构体指针变量进行成员输出的效果是一样的。

使用结构体指针变量需要注意以下方面。

(1) 使用结构体指针变量访问结构体变量成员的格式有两种方式：

```
(*结构体指针变量名).成员名;
结构体指针变量名->成员名;
```

当采用第一种方式时，(*结构体指针变量名)的括号必须保留，这是因为“*”运算符的优先级低于“.”，如果省略括号，上例中(*p).name 就变成*p.name，等价于*(p.name)。第二种方式为指针专用的指向运算符。这两种方式的访问效果是相同的。

(2) 结构体指针变量在程序中必须指向某一结构体变量，然后才能使用指针。

给结构体变量的各个成员赋值。如以下使用方式不正确：

```
Employee *p;
(*p).employeeid = "20100002";
```

当定义了该结构体指针后，指针的类型虽然是结构体类型，但是还没有完成初始化，还没有

在内存中分配结构体变量所占的内存空间，应当首先让结构体指针变量 p 指向某一已经定义的结构体变量，例如：

```
Employee employee1,*p;
p=&employee1;
(*p).employeeid="20100002";
```

C++语言的指针使用非常灵活，如果使用恰当可以提高程序的执行效率。在例 7.3 中使用结构体变量作为函数参数，这种参数传递方式为值传递，即当传递参数时，会把参数值复制一份，当该结构变量很大时，会使程序所占内存增加，降低程序执行效率。如果改成使用结构体指针作为参数，函数参数传递方式为地址传递，不需要复制结构体变量，从而提高效率。

【例 7.6】 把例 7.3 改成使用结构体变量指针作为函数参数。

```
#include <iostream>
#include <string>
using namespace std;
struct Date             //定义结构体类型 Date
{
  int month;
  int day;
  int year;
};
struct Employee         //定义结构体类型 Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
   Date worktime;
} employee1={"20100001", "王军", 25, 2100.5, 2, 20, 2010};

void printemployee(Employee *p)  //参数为 Employee 结构类型变量
{
    cout<<"员工"<<(*p).name<<"的基本信息为:"<<endl;
    cout<<"员工号:"<<(*p).employeeid<<endl;
    cout<<"姓名:"<<(*p).name<<endl;
    cout<<"年龄:"<<(*p).age<<endl;
    cout<<"工资:"<<(*p).wage<<endl;
    cout<<"参加工作时间:"<<(*p).worktime.year<<"-"
    <<(*p).worktime.month<<"-"
    <<(*p).worktime.day<<endl;
}
int main()
{
    Employee employee2, *p;
    p = &employee2;
    employee2.employeeid = "20100002";
    employee2.name = "李明";
    employee2.age = 28;
```

```
    employee2.wage = 3500.5;
    employee2.worktime.month = 3;
    employee2.worktime.day = 15;
    employee2.worktime.year = 2010;
    printemployee(p);
    employee2 = employee1;
    printemployee(p);
    return 0;
}
```

将输出如下的结果：

员工李明的基本信息为：
员工号：20100002
姓名：李明
年龄：28
工资：3500.5
参加工作时间：2010-3-15
员工王军的基本信息为：
员工号：20100001
姓名：王军
年龄：25
工资：2100.5
参加工作时间：2010-2-20

使用结构体指针作为函数参数可以提高程序运行效率，但是指针操作较为复杂，容易出错。在上例中，可以使用结构体引用变量作为函数参数传递，在提高效率的同时可以使程序保持原来的写法。

【例 7.7】 把例 7.3 改成使用结构体引用变量作为函数参数。

```
#include <iostream>
#include <string>
using namespace std;
struct Date         //定义结构体类型 Date
{
  int month;
  int day;
  int year;
};
struct Employee     //定义结构体类型 Employee
{
   string employeeid;
   string name;
   int age;
   double wage;
   Date worktime;
} employee1={"20100001", "王军", 25, 2100.5, 2, 20, 2010};

void printemployee(Employee &employee) //参数为 Employee 结构类型变量
```

```
{
    cout<<"员工"<< employee.name<<"的基本信息为:"<<endl;
    cout<<"员工号:"<< employee.employeeid<<endl;
    cout<<"姓名:"<< employee.name<<endl;
    cout<<"年龄:"<< employee.age<<endl;
    cout<<"工资:"<< employee.wage<<endl;
    cout<<"参加工作时间:"<< employee.worktime.year<<"-"
    << employee.worktime.month<<"-"
    << employee.worktime.day<<endl;
}
int main()
{
    Employee employee2;
    employee2.employeeid = "20100002";
    employee2.name = "李明";
    employee2.age = 28;
    employee2.wage = 3500.5;
    employee2.worktime.month = 3;
    employee2.worktime.day = 15;
    employee2.worktime.year = 2010;
    printemployee(employee2);
    employee2 = employee1;
    printemployee(employee2);
    return 0;
}
```

将输出如下的结果：

员工李明的基本信息为：
员工号：20100002
姓名：李明
年龄：28
工资：3500.5
参加工作时间：2010-3-15
员工王军的基本信息为：
员工号：20100001
姓名：王军
年龄：25
工资：2100.5
参加工作时间：2010-2-20

可以看到，采用引用变量作为函数参数时，程序的执行效果和使用指针变量作为函数参数是一样的，但是程序可读性较好。

7.1.7　结构体与链表

链表(linked list)是重要的数据结构之一。链表一般由结构体变量构成，链表中存储的信息保存在结构体变量的成员中。表示链表的结构体类型的定义方式如下：

```
struct Employee
{
  string name;
  double wage;
  Employee *next;
};
```

定义了结构体类型 Employee，它有三个成员，前两个表示员工姓名和工资，最后一个成员是结构体指针变量 next，它所指向的结构体变量也是 Employee 类型。图 7-2 显示了链表结构。

在图7-2中，最左边有一个指针变量，指向链表中的第一个结构体变量，表示链表的开始，图中用 head 来表示。每个块状部分代表链表中的一个基本组成部分，一般称做结点(node)。在该链表结构中，每个结点由三部分组成，分别对应结构体中的三个成员，其中最后一个成员为指针，指针的值为下一个结构体变量的起始地址，即指向链表中的下一个结点。链表中最后一个结点的最后一个成员为一个空指针，表示链表结束。在 C++中由 NULL 来表示。

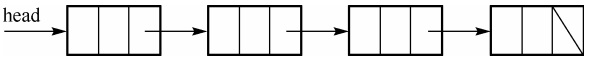


图 7-2 简单链表结构

【例 7.8】 建立一个表示员工信息的简单链表。

```
#include <iostream>
#include <string>
using namespace std;
struct Employee
{
      string name;
      double wage;
      Employee *next;
 };
int main()
{
Employee employee1,employee2,employee3;
      Employee *head, *p;
      employee1.name = "王军";
      employee1.wage = 2100.5;
      employee1.next = &employee2;
      employee2.name = "李明";
      employee2.wage = 3500.5;
      employee2.next = &employee3;
      employee3.name = "刘云";
      employee3.wage = 2810.5;
      employee3.next = NULL;
      head = &employee1;
      p = head;
      while(p!=NULL)
      {
        cout<<"员工"<<p->name<<"的工资为"<<p->wage<<endl;
        p = p->next;
```

```
        }
        return 0;
    }
```

将输出如下的结果：

员工王军的工资为 2100.5
员工李明的工资为 3500.5
员工刘云的工资为 2810.5

在例 7.8 中，首先定义 Employee 结构体类型的 3 个变量。通过对变量的各个成员的赋值过程完成链表的构建。其中“链接”不同结点的过程由以下语句实现：

```
employee1.next = &employee2;
employee2.next = &employee3;
employee3.next = NULL;
```

第一个结点的成员 next 指针保存了第二个结点在内存中的起始地址，第二个结点的成员 next 指针保存了第三个结点在内存中的起始地址，最后一个结点由于不指向其他结点，所以定义其 next 指针的值为 NULL。这样三个就“链接”在一起。程序中定义了用于指向第一个结点的 head 指针和用于循环遍历整个链表的 p 指针。当访问链表中的每一个结点时，首先把 head 指针的值赋给 p，即由 p 指向链表中的第一个结点，这时可以访问第一个结点中成员的值，然后把第一个结点的成员 next 的值赋给 p，即 p 此时指向链表中的第二个结点，这时可以访问第二个结点中成员的值。最后把第三个结点的成员 next 的值赋给 p，此时 p 的值为 NULL，遍历结束。

链表和结构体数组都可以容纳大量结构体变量，链表和结构体数组的主要区别如下。

（1）结构体数组的各个元素在内存中是连续的，而链表中的各个结点在内存中可以不连续，其访问方式是通过存储在结点中的指针来进行的。

（2）数组一般需要在定义时指定数组中元素的数目，而链表则可以动态添加、删除新的结点，所以链表不需要指定结点数目，使用比较灵活。实际应用中，为了容纳足够多的元素，定义数组时其数目一般需要有一定的余量，即数组中的元素一般很少能够填满数组，所以当需要存储和处理的信息比较多时，链表可以节省内存，尤其是结构体变量很大，而其个数也很多时更加明显。

7.2　联合

7.2.1　联合的定义

联合也是一种自定义数据类型，联合中可以包括多种不同类型的变量作为成员，但这些不同数据类型的成员是“共享”同一段内存的。所谓“共享”，指当一个联合变量被声明之后，在某一时刻该联合变量中只能有一个数据类型的成员在内存中存储，所以也称联合为共用体。联合可以理解为一种特殊的结构体，与结构体不同的是，结构体变量是结构体中多个成员的集合，其长度一般为结构体中各个成员长度之和(C++编译器为了计算方便一般会进行长度取整，所以结构体变量的长度可能不完全等于结构体中各个成员的长度之和)；而联合变量在某一时刻只有一个成员起作用，联合变量的长度为其长度最大的成员的长度。

定义联合类型的一般形式为：

```
union 联合类型名
{
数据类型名1  成员1;
数据类型名2  成员2;
…
数据类型名n  成员n;
};
```

例如，定义一个联合类型，其中两个成员分别为整型和双精度数：

```
union unknownvar
{
    int asint;
    double asdouble;
};
```

定义联合类型时，其中成员的类型可以为 C++的基本数据类型和自定义类型。如以下联合的定义是允许的：

```
union unknownvar
{
    int asint;
    struct date
    {  int month;
       int day;
       int year;
    } asdate;
};
```

需要注意的是，当联合中使用自定义类型对成员定义时，该自定义类型不能有构造函数、析构函数和拷贝构造函数(这三种函数用做类的成员函数，在第 8 章将详细介绍)。比如在联合类型中就不能使用 string 来定义成员(string 是 C++标准库中定义的类，用来表示字符串，拥有构造函数、析构函数和拷贝构造函数)。如定义以下联合类型：

```
union unknownvar
{
    int asint;
    string asstring;
};
```

编译器将给出如下错误提示：

error C2621：成员"unknownvar::asstring"（属于联合"unknownvar"）具有复制构造函数

7.2.2 联合变量的定义

联合类型的变量声明方式与结构变量类似，主要有以下三种方式：

(1) 先定义联合类型再声明结构体变量。

使用 7.2.1 中定义的联合类型声明变量可以使用以下形式：

```
unknownvar a, b;
```

表示使用 unknownvar 联合类型声明了两个变量 a 和 b。这种声明方式与 C++标准类型变量声明是一样的。

(2) 定义联合类型的同时声明变量。

```
union unknownvar
{
    int asint;
    double asdouble;
} a, b;
```

(3) 直接声明联合类型变量

```
union
{
    int asint;
    double asdouble;
} a, b;
```

与结构体类似，联合类型变量也可以在声明时进行初始化。如果使用第一种联合变量声明方式，其初始化过程如下：

```
unknownvar  a={10};
```

如果使用第二种结构体声明方式，其初始化过程如下：

```
union unknownvar
{
    int asint;
    double asdouble;
} a={10};
```

需要注意的是，在 VC++ 2005 中初始化的时候只能对 union 的第一个成员赋初值，对 unknownvar 类型变量初始化操作如果写成以下形式：

```
unknownvar  a={10.5};
```

则编译器会给出如下警告提示：

warning C4244: “初始化”：从“double”转换到“int”，可能丢失数据

从该警告可以得知，编译器仍然是对 unknownvar 类型的第一个成员赋初值，同时进行了从“double”到“int”隐式类型转换。

当联合类型进行初始化操作时，如果对该类型的第二个成员赋初值，例如：

```
union unknownvar
{
    int asint;
    struct date { int month;  int day;  int year; } asdate;
} a={{8,10,2010}};
```

则编译器会给出以下错误提示：

error C2078：初始值设定项太多

如果调换一下 unknownvar 类型中成员定义的顺序，再进行如下初始化操作，则编译器不

会给出错误提示。

```
union unknownvar
{
    struct date { int month;  int day;  int year; } asdate;
    int asint;
} a={{8,10,2010}};
```

7.2.3 联合变量的引用

当使用联合类型声明了联合变量后，就可以使用该变量。引用联合变量时只能对联合变量的成员进行操作。与结构体变量成员的引用类似，联合变量成员的引用格式为：

联合变量名.成员名

由于联合变量中的成员在某一时刻只能有一个在起作用，所以在引用时需要注意当前起作用的是哪一个成员，如果把该成员当成其他成员引用，编译器可能不会给出错误提示，但是结果可能会不正确。

【例 7.9】 使用联合进行两个数的比较。

```
#include<iostream>
using namespace std;
union unknownvar
{
    int asint;
    double asdouble;
};
bool compare(unknownvar a,unknownvar b, int flag)
{
    if(flag==0)
    {
    if(a.asint>b.asint) return true;
    else return false;
    }
    if(flag==1)
    {
    if(a.asdouble>b.asdouble) return true;
    else return false;
    }
}
int main( )
{
    unknownvar a,b,c,d;
    a.asint=10;
    b.asint=15;
    c.asdouble=6.3;
    d.asdouble=6.2;
    cout<<"整型和双精度数的长度分别为"<<sizeof(int)<<"和"<<sizeof(double)<<endl;
    cout<<"联合变量 a 的长度为"<<sizeof(a)<<endl;
    cout<<"联合变量 c 的长度为"<<sizeof(c)<<endl;
```

```
    if(compare(a,b,0))
    cout<<a.asint<<"大于"<<b.asint<<endl;
    else cout<<a.asint<<"小于"<<b.asint<<endl;
    if(compare(a,b,1))
    cout<<c.asdouble<<"大于"<<d.asdouble<<endl;
    else cout<<c.asdouble<<"小于"<<d.asdouble<<endl;
    return 0;
}
```

输出结果为：

整型和双精度数的长度分别为 4 和 8
联合变量 a 的长度为 8
联合变量 c 的长度为 8
10 小于 15
6.3 大于 6.2

在例 7.9 中，首先声明 unknownvar 联合类型，其中包括两个成员，一个用来存储整数，一个用来存储双精度数。然后声明函数 compare 来比较两个数的大小，其中有三个参数，前两个参数为 unknownvar 联合类型变量，最后一个参数 flag 作为选择标志。如果 flag 的值取 0，则表示引用联合类型变量的整数成员；如果 flag 的值取 1，则表示引用联合类型变量的双精度数成员。主函数中声明了 4 个 unknownvar 联合类型变量，使用这 4 个变量进行比较操作。从运行结果可以看出，尽管整型和双精度数的长度分别为 4 和 8，联合变量 a 和 c 尽管使用不同的成员，但它们的长度均为所有成员中长度的最大值 8。

7.3　枚举类型

在程序设计过程中，经常会出现某个变量只有几种可能取值的情况，例如，交通信号灯的颜色有红、黄和绿三种不同颜色，一周中的某一天只能取周一至周日中的某个值，一年中的某个月份只能为 1 月至 12 月之间的某个值。在 C++中可以定义枚举类型，使用枚举来保存一组用户定义的值。与结构体和联合一样，枚举也是一种自定义数据类型，使用时必须先声明一个枚举类型，然后才能使用该枚举类型定义变量。

声明枚举类型的一般形式为：

```
enum 枚举类型名 {枚举成员表列};
```

以下为一些枚举类型声明的例子：

```
enum color{red,yellow,green};
enum day{sun,mon,tue,wed,thu,fri,sat};
enum season{spring,summer,fall,winter};
```

使用枚举类型定义变量的一般形式为：

枚举类型名　枚举变量表列;

对以上声明的枚举类型，可以做如下变量定义：

```
color light;
day today,tommorow;
```

```
season oneseason;
```

对枚举类型，需要注意以下几点。

(1) 声明枚举类型并使用该类型进行变量定义时，变量只能被赋值为枚举类型中的某个枚举成员。

如对于 color 枚举类型，light 变量的值只能为 red、yellow 和 green 其中之一，其赋值语句可以写成如下形式：

```
light=yellow;
```

(2) 在 C++系统枚举类型中的枚举成员的值是用整数来实现的。在枚举成员表列中的第一个成员的值为 0，第二个为 1，依次类推。

对于以下语句：

```
light=yellow;
cout<<light<<endl;
```

程序会输出 1。

可以在枚举类型声明时指定某个成员的值，例如：

```
enum season{ spring,summer=3,fall,winter };
season a,b,c,d;
a=spring; b=summer; c=fall; d=winter;
cout<<a<<" "<<b<<" "<<c<<" "<<d<<endl;
```

上述语句执行的结果为 0 3 4 5。

从结果可以看出，当给枚举类型中的某个成员赋一个整数值时，如果该成员为第一个成员，则后边的成员的值按照顺序依次加 1；如果该成员不是第一个成员，则第一个成员开始直到该成员前一个成员按照默认赋值的规则进行赋值，即第一个成员的值为 0，以后依次加 1。

(3) 由于枚举变量的值为整数，所以枚举值可以用来做判断比较，如以下语句是正确的：

```
if(light==green) {cout<<"绿灯"<<endl;}
if(ligth>red) {cout<<"黄灯或绿灯"<<endl;}
```

C++中使用枚举类型主要目的是提高程序的可读性，枚举变量经常用于多个条件的判断，实际应用中枚举经常与 switch 或 if 等判断语句结合使用。例如：

```
void prinfcolor(int x)
{
    switch(x)
{
    case 0:
      cout<<"红色"<<endl;
      break;
    case 1:
      cout<<"黄色"<<endl;
      break;
    case 2:
      cout<<"绿色"<<endl;
```

```
        break;
    default:
        cout<<"红黄绿以外的其他颜色"<<endl;
}
}
```

上述函数中根据参数 x 的值来判断输出，其中的每个分支选项用 0、1、2 来表示，具体含义不是很明确，可能造成程序理解困难。如果使用枚举类型 color，程序的可读性就会大大提高，例如：

```
void prinfcolor(color x)
{
    switch(x)
    {
    case red:
      cout<<"红色"<<endl;
      break;
    case yellow:
      cout<<"黄色"<<endl;
      break;
    case green:
      cout<<"绿色"<<endl;
      break;
    default:
      cout<<"红黄绿以外的其他颜色"<<endl;
      }
}
```

【例 7.10】 使用枚举类型。

```
#include<iostream>
#include<string>
using namespace std;
enum weekday{mon=1,tue,wed,thu,fri};
void prinfcourse(weekday day)
{
   switch(day)
   {
      case mon:
        cout<<"英语"<<endl;
        break;
      case tue:
        cout<<"数学"<<endl;
        break;
      case wed:
        cout<<"语文"<<endl;
        break;
      case thu:
        cout<<"体育"<<endl;
```

```
            break;
        case fri:
            cout<<"物理"<<endl;
            break;
        default:
            cout<<"输入的日期不正确! "<<endl;
        }
}
int main()
{
    weekday searchday;
    int searchnum;
    cout<<"请输入查询周几的课表(1 代表周一，2 代表周二……5 代表周五)："<<endl;
    cin>>searchnum;
    searchday=(weekday)searchnum;
    prinfcourse(searchday);
    return 0;
}
```

在例 7.10 中，声明了一个 weekday 枚举类型，并让其成员从 1 开始赋值。通过用户的输入来查询相应的课表。枚举变量 searchday 只能赋值为枚举 weekday 中的成员，但是用户的输入为数字，所以程序中使用了 searchday=(weekday)searchnum;做了转换，把用户输入的数字首先转换为枚举值，然后调用函数执行查询。

7.4 结构体与联合应用实例

联合类型变量的所有成员共享同一块内存空间，在某一时刻只能有一个成员起作用，为了防止引用联合变量成员时出错，可以引入“标志”，利用标志的值来指示应该引用联合变量的哪个成员，这种情况下可以把联合变量和“标志”放在一个结构体中，当使用结构体变量时由里边的标志成员来确定如何引用联合变量的成员。

【例 7.11】 使用结构体和联合进行员工管理。

假设一个企业中员工分为两类，一类员工按所生产产品件数来计算工资，一类员工按工作小时数来计算工资。这两类员工都可以声明为一个结构体，然后在结构体中定义一个标志成员来区分不同员工。由于一种员工只能有一个工资计算方法，所以在员工结构体中声明一个联合类型，成员为产品件数和工作小时数，在计算工资时就可以根据标志成员取值的不同调用不同的计算函数。

```
#include <iostream>
#include <string>
using namespace std;
struct Employee
{
    string employeeid;
    string name;
    int age;
```

```
    float wage;
    union
    {
        int productnum;
        double workhours;
   };
   int flag;
};
Employee inputemployee(Employee &employee)
{
cout<<"请输入员工信息"<<endl;
cout<<"员工号:";
cin>>employee.employeeid;
cout<<"员工姓名:";
cin>>employee.name;
cout<<"员工年龄:";
cin>>employee.age;
cout<<"员工类别(输入 0 表示计件员工，输入 1 表示计时员工):";
cin>>employee.flag;
if(employee.flag==0)
{
       cout<<"产品件数:";
       cin>>employee.productnum;
}
if(employee.flag==1)
{
       cout<<"工作小时数:";
       cin>>employee.workhours;
}
if(employee.flag==0)
{
       employee.wage=5*employee.productnum;    //假设每件产品的工资为 5 元
}
if(employee.flag==1)
{
       employee.wage=50*employee.workhours;    //假设每小时的工资为 50 元
}
return employee;
}
void printemployee(Employee employee)
{
   cout<<"员工"<<employee.name<<"的基本信息为:"<<endl;
   cout<<"员工号:"<<employee.employeeid<<endl;
   cout<<"姓名:"<<employee.name<<endl;
   cout<<"年龄:"<<employee.age<<endl;
   cout<<"工资:"<<employee.wage<<endl;
}
int main( )
```

```
{
    int i;
    Employee employee[10];
    for(i=0;i<10;i++)
    {
          inputemployee(employee[i]);
    }
    for(i=0;i<10;i++)
    {
          printemployee(employee[i]);
    }
    return 0;
}
```

在例 7.11 中，首先定义了结构体类型 Employee，该结构体中包含一个联合类型，该联合类型没有名字，在 C++中称为匿名联合。使用匿名联合时，可以把该联合中的成员作为外层结构体的成员来直接引用。如以下语句表示假设每件产品付给员工工资 5 元时员工工资的计算过程。

```
employee.wage=5*employee.productnum;
```

本 章 小 结

结构体和联合是 C++语言中重要的自定义数据类型。结构体把一组变量“组合”起来，从而形成一个具有多种属性的记录。与使用标准数据类型一样，可以使用结构体声明变量，使用结构体数组、结构体指针，使用结构体作为参数等。当结构体和指针结合使用时，可以生成重要的数据结构——链表。同时，结构体也是 C++面向对象程序设计中“类”的一种表现形式。联合是多种数据类型成员“共享”同一段内存的自定义数据类型，某一时刻一个联合变量中只有一个成员在起作用，联合变量的长度为其所有成员中长度最大成员的长度。枚举类型用来表示一个变量的几种不同取值，C++中枚举成员的值用整数表示。

习 题 7

一、填空题

1. 结构体变量中的成员的引用形式为_____。
2. 设有程序 struct node{int x; int y;} *p;，且 p 已经指向一个 node 变量，指针 p 指向的 node 变量中的成员 y 的表示方法为 _____。
3. 设有程序 union node{double x; int y;} node1={10.5};，则 node1.y 的值为_____。
4. 设有程序 enum flag{a,b=2,c,d}; flag flag1=c;，则 flag1 的值为_____。

二、分析题

1. 设有以下结构体和联合类型说明及变量定义，试分析变量 stu1、data 所占字节数。

```
struct student
```

```
{
char name[10];
double ave;
struct Date
{
   int month;
   int day;
   int year;
} birthday;
} stu1;
union Data
{
   char ch[4];
   double d;
} data;
```

2. 分析以下程序片段存在的错误并改正。

```
struct student
{
   string id;
   string name;
   struct date{
   int month; int day; int year;
   }
   date birthday;
} stu1={"20100001","胡明", 10,15,1988};
```

三、编程题

设计一个学生链表，设每个学生的数据包括学号、姓名、年龄和入学成绩，程序要求：

(1) 用户从键盘输入学生数据完成链表的创建过程；

(2) 用户输入学生学号查找入学成绩。

第8章　类 与 对 象

C++语言是一门混合型程序设计语言，一方面可以使用 C++进行面向过程的程序设计，把一个复杂的程序划分为多个模块，每个模块又可以划分为多个子模块，通过这种“自顶向下，逐步分解”的方法把一个复杂的问题变成多个小问题来实现，而每个小问题可以设计为C++的函数，最后把这些函数有机组合在一起，就可以解决复杂程序的设计问题。另一方面C++又是一门重要的面向对象的程序设计语言。在面向对象程序设计方法中构成模块的基本单元是“类”和“对象”，通过设计类的属性和方法把数据和函数封装在一起，由类来完成单位模块的功能，通过设计不同类的相互关系来实现类和对象间合作，从而协作完成复杂问题的求解。

8.1　类的概念与定义

8.1.1　面向对象程序设计概述

面向对象程序设计(Object-Oriented Programming)是C++程序设计语言的重要程序设计方法，使用面向对象程序设计方法可以完成更复杂、规模更大的程序。与面向对象程序设计相对应的是面向过程程序设计，前面介绍的程序设计方法主要是面向过程的，在面向过程的程序设计中，程序的核心部分是函数。我们把一个复杂的程序过程分解为不同的函数，每个函数解决一个子问题，最后由这些函数协作来完成整个程序过程。这样就把一个大的复杂的问题变成对若干个小的简单问题的求解，这种方法称为“自顶向下，逐步求精”。下面通过一个具体的例子来理解函数作为模块进行程序的方法。

【例8.1】 使用函数作为模块的程序设计方法。

假设某个企业中员工岗位分为两类，一类为技术型，一类为管理型。现在企业需要根据员工类型和员工参加工作时间计算津贴。假设企业员工信息存储在一个结构数组中，该结构包括了员工的编号、姓名、类型、参加工作时间等信息，要求根据输入的员工编号，查找出该员工，然后根据员工类型和参加工作时间计算津贴。

```
#include <iostream>
#include <string>
using namespace std;
struct Date
{
   int month;
   int day;
   int year;
};
struct Employee
{
   string employeeid;
```

```
    string name;
    int kind; //员工类型标识，技术类为 1，管理类为 2
    Date worktime;
};
//查找员工
int SearchEmployee(Employee *p,int n,string employeeid)
{
    int i;
    for(i=0;i<n;i++)
    {
    if((*(p+i)).employeeid==employeeid)
        return i;
    }
        return -1;
}
double ComputeAllowance(int kind,int workyear)
{
if(kind==1)
{
    if(workyear<=2000)
         return 10000;
    else
         return 5000;
}
else
{
    if(workyear<=2000)
         return 6000;
    else
         return 3000;
}
}
void PrintAllowance(Employee *p,int i)
{
if (i==-1)
{
         cout<<"没有与输入员工编号相匹配的员工记录"<<endl;
}
else
{
         cout<<"员工"<<(*(p+i)).name<<"的津贴为:"
<<ComputeAllowance((*(p+i)).kind,(*(p+i)).worktime.year)<<endl;
}
}
int main( )
{
int i=0;
string searchid;
```

```
    Employee employee[4]={
        {"19900001","王军", 1,3,15,1990},
        {"20000001","李明", 2,10,8,2000},
        {"20050001","刘云", 1,5,20,2005},
        {"20070001","赵涛", 2,8,18,2007}};
    cout<<"请输入员工编号:";
    cin>>searchid;
    i=SearchEmployee(employee,4,searchid);
    PrintAllowance(employee,i);
    return 0;
    }
```

在例 8.1 中，程序的要求是由用户输入员工编号，在结构体数组中找到该员工，根据该员工的参加工作时间和员工类型进行津贴计算(为了简单起见，津贴计算部分只根据 2000 年前后参加工作和两种不同的员工类别进行处理)并输出。把整个程序过程分成三个部分：查找员工、计算津贴和打印输出。这三个部分分别由三个函数来实现，即程序中的 SearchEmployee、ComputeAllowance 和 PrintAllowance 三部分。首先由 SearchEmployee 函数根据用户输入的员工编号得到该员工在数组中的下标，根据下标获取员工的类型和参加工作时间，然后由 ComputeAllowance 函数通过员工类型和参加工作时间计算得出津贴，最后由 PrintAllowance 函数输出该员工的津贴。函数与数据及函数与函数的调用关系如图8-1所示：

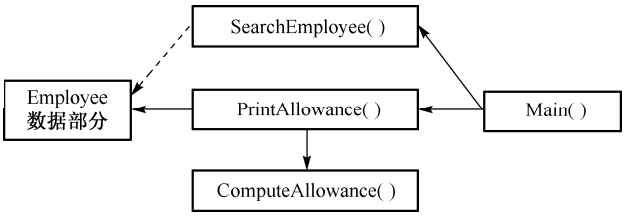


图 8-1　例 8.1 中数据与函数关系

其中实线代表函数间的调用关系，虚线代表函数对数据的使用关系，箭头代表了函数调用或函数使用数据的方向。从程序中可以看到 SearchEmployee 和 PrintAllowance 两个函数的参数都包含结构体 Employee 的指针变量，即这两个函数直接调用了数据 Employee。函数与数据，以及函数与函数之间依赖关系的紧密程度称为“耦合度”，“耦合度”越高说明函数与数据或函数与函数之间的依赖性越强，反之则依赖性越差。在本例中，SearchEmployee 和 PrintAllowance 两个函数通过参数直接访问数据 Employee，说明它们之间是紧密耦合的。

使用函数作为模块进行面向过程的程序设计主要存在以下几个缺点:

(1) 数据与操作它们的函数分开，数据的内容可能会被不相关的函数“误访问”。而且当数据与函数为紧密耦合关系时，如果数据的形式和内容发生变化，操作它们的函数往往也要跟着变化。

(2) 把一个较大规模程序分解成多个函数模块，这些模块间一般是具有一定耦合关系的(尽管可以通过良好的设计实现最低程度的耦合)，此时如果上层调用函数发生变化，很有可能会引起下层被调用函数的变化，从而产生串联改变。在图8-1中函数只有 4 个，如果程序规模很大，函数的数目很多，调用关系很复杂，如图8-2所示，将会使串联改变变得难以控制。

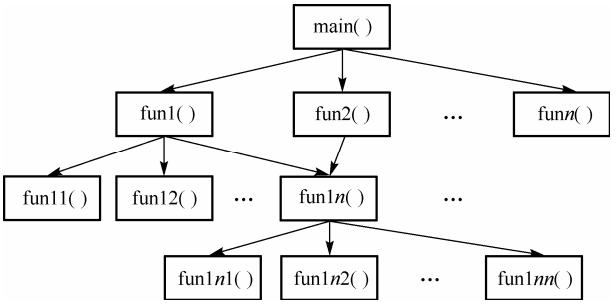


图 8-2　函数的复杂层级关系

(3) 尽管可以通过良好的设计尽量消除函数间的依赖关系，但是依赖关系很难完全避免，具有依赖关系的函数重用性较差，很难移植到其他程序中。

上述情况均会导致程序的理解和维护难度增大，所以使用函数作为模块进行面向过程程序设计时，程序超出一定的规模会极大地增加开发和维护成本。

面向对象程序设计方法主要着眼于解决面向过程程序设计在开发大型程序时所面临的一系列缺点：数据与操作分离、函数的串联改变、模块的可重用性差等。C++引入了“类”和“对象”的概念来支持面向对象程序设计，此时构成程序的基本模块是类和对象。类把数据和相关的操作捆绑在一起，形成一个整体。类中的数据称为**数据成员**，类中的操作称为**成员函数**。面向对象程序设计的主要特点有：抽象、封装、继承和多态。

(1) 抽象

抽象表示同一类事物的本质。比如“学生”是一种抽象，它是对现实世界中具体学生个体的一种归纳。

在面向对象程序设计中，类和对象就是抽象与具体的关系。类是对象的抽象，对象是类的具体实例。

(2) 封装

类和对象把数据和相关的操作放在一起，作为一个整体对外界提供服务。当外部程序代码使用对象时，并不需要全部了解对象内部所有的数据和操作代码，只需要通过一个明确定义的接口来控制。就像我们平时看电视只需要操作遥控器的按钮，而不需要了解电视内部的元器件和电路控制原理。封装有两层含义：一是把数据和相关操作捆绑在一起，使其成为一个单独的整体；二是隐藏内部实现细节，把部分数据和操作隐藏起来，只提供适当的接口供外界使用。封装降低了外部程序代码操作对象的复杂程度，同时避免了内部数据被外部程序代码因“误操作”而被修改。

(3) 继承

继承为代码重用提供了一种解决机制。假设有一个“员工”类，里边包括员工编号、姓名、年龄等属性和计算工资等方法。现在需要定义一个“技术类员工”类，“技术类员工”是一种“员工”，里边也包括编号、姓名、年龄等属性和计算工资的方法，另外还可能具有一些特殊的属性，如技术领域、技术职称等。这种情况下，就可以让“技术类员工”类从“员工”类继承基本属性，然后再添加特殊属性。这种机制会大大提高程序的编写效率，对大型软件的开发具有很重要的意义。

(4) 多态

多态是面向对象程序设计的一个重要特征。多态可以理解为：给多个不同的对象下达同

一个指令，不同的对象在接受时会产生不同的行为。比如企业中不同的员工工作岗位是不一样的，工作方式也不一样。企业经理向所有员工下达工作的指令，每个人接受到指令后会按照自己方式在自己的岗位上工作。C++中多态是在继承的基础上实现的。

在 C++中，面向对象程序设计主要是通过类(class)来实现，也可以通过给结构体(struct)增加成员函数的方法来实现。第7章中设计的结构体中只包括了数据成员，C++允许在结构体中增加成员函数，这样就把数据与相关的操作结合在一起，从而实现了 C++面向对象的封装。

【例 8.2】 通过给结构体增加成员函数来实现封装。

```
#include <iostream>
#include <string>
using namespace std;
struct Date
{
   int month;
   int day;
   int year;
};
struct Employee
{
   string employeeid;
   string name;
   int kind; //员工类型标识，技术类为 1，管理类为 2
   Date worktime;
   bool SearchEmployee(string searchid)
   {
    if(employeeid==searchid)
        return true;
    else
        return false;
   }
   double ComputeAllowance( )
   {
    if(kind==1)
    {
        if(worktime.year<=2000)
            return 10000;
        else
            return 5000;
    }
    else
    {
        if(worktime.year<=2000)
            return 6000;
        else
            return 3000;
    }
```

```
  }
  void PrintAllowance()
  {
    cout<<"员工"<<name<<"的津贴为:"<<ComputeAllowance()<<endl;
  }
};
int main()
{
    string searchid;
    Employee employee[4]={
    {"19900001","王军", 1,3,15,1990},
    {"20000001","李明", 2,10,8,2000},
    {"20050001","刘云", 1,5,20,2005},
    {"20070001","赵涛", 2,8,18,2007}};
    cout<<"请输入员工编号:";
    cin>>searchid;
    for(int i=0;i<4;i++)
    {
    if(employee[i].SearchEmployee(searchid))
    {
        employee[i]. PrintAllowance ();
        return 0;
    }
    }
    cout<<"未找到与员工编号"<<searchid<<"相匹配的员工记录"<<endl;
    return 0;
}
```

在例 8.2 中，把 SearchEmployee、ComputeAllowance、PrintAllowance 三个函数放在结构体内，使它们成为结构体 Employee 的成员函数，作为结构体的成员函数，它们可以直接操作结构体内的数据成员，所以这些函数不再需要包含 Employee 类型的函数参数。当结构体的名称发生变化时，只要内部数据成员没有变化，其成员函数不需要做改变。需要注意的是，这个例子中只是简单地把数据成员和成员函数捆绑在了一起，并没有对外部做内部信息的隐藏，外部程序代码仍然可以访问内部的所有成员(包括数据成员和成员函数)。即该例中实现的是封装的第一层含义。

8.1.2　类的声明

C++中，任何变量的使用都需要先声明其数据类型，比如使用一个整型的数据 i，首先需要使用以下语句进行定义：

```
int i;
```

如果变量的类型为自定义类型，那么首先需要声明该自定义类型，然后再使用该自定义数据类型进行变量的定义。假设有一个结构体类型 Date，使用 Date 定义变量需要如下进行：

```
struct Date
{
   int month;
```

```
    int day;
    int year;
};
Date birthday;
```

作为C++语言中的自定义类型，类在定义变量(对象)前同样需要先声明。其声明的方式为：

```
class 类名
{
   private:
     私有的数据成员;
     私有的成员函数;
   public:
     公有的数据成员;
     公有的成员函数;
   protected:
     保护的数据成员;
     保护的成员函数;
};
```

可以声明如下员工类：

```
class Employee
{  private:
     string employeeid;
     string name;
     int age;
   public:
     void display( )
     {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
     }
};
```

class 为 C++中定义类的关键字，类名可以为 C++中所有合法的标识符，private、public 和 protected 为成员访问限定符，用来表示类中数据成员和成员函数的访问特征。这三种访问控制方式分别为：

(1) 类中所有 private 成员(包括数据成员和成员函数)只能被本类中的成员函数访问。外部程序代码访问类中私有成员是非法的(友元函数和友元类例外)。

(2) 类中所有的 public 成员(包括数据成员和成员函数)既可以被本类中的成员函数访问，也可以被外部的程序代码访问。

(3) protected 限定符一般用于类的继承，一个类的子类称为派生类，类的 protected 成员不能被类外部程序代码访问，但是可以被派生类的成员函数访问。

一个成员访问限定符在遇到另一成员访问限定符之前，对它后边所有的成员均有效，而且在一个类中可以按照任意顺序放置任意数目的 private、public 和 protected 限定符。如以下声明是合法的：

```
class Employee
{  public:
      string name;
      int age;
   private:
      string employeeid;
   public:
      void display( )
   {
      cout<<"编号:"<< employeeid<<endl;
      cout<<"姓名:"<< name <<endl;
      cout<<"年龄:"<< age <<endl;
   }
};
```

以上类的声明首先包括两个公有数据成员，然后是一个私有数据成员，最后是一个公有成员函数，一般情况在声明一个类时往往把使用相同访问控制的成员写在一个限定符下。

8.1.3　类的成员函数

当一个函数属于某个类时，称其为类的成员函数，类的成员函数和 C++语言中的普通函数在定义时是一样的，只是现在该函数是这个类的成员。前边所定义的成员函数声明和定义都是写在类的大括号内。也可以把成员函数的声明和定义分开，把声明部分写在类内，把定义部分写在类的外部。例如：

```
class Employee
{  private:
      string employeeid;
      string name;
      int age;
   public:
      void display( )
   };
void Employee::display( )
{
      cout<<"编号:"<< employeeid<<endl;
      cout<<"姓名:"<< name <<endl;
      cout<<"年龄:"<< age <<endl;
};
```

在以上类的定义中，类内只保留了成员函数 display 的声明，而 display 函数的定义写在类外。由于不同的类可以具有同名的成员函数，所以当在类外进行成员函数的定义时，需要在函数名前边加上类名，并使用作用域限定符“::”指示该函数是属于哪个类的。在 VC++ 2005 中，当一个类规模较大时，可以让一个类的声明中只保留数据成员和成员函数的原型，然后把该声明放在一个扩展名为“.h”的头文件中，把类中成员函数的定义部分放在扩展名为“.cpp”文件中，同时使用 include 在该 cpp 文件中引用类的声明文件。假设把以上 Employee 类的声明和类成员函数的声

明和定义部分分别存放在 EmployeeManagement.h 和 EmployeeManagement.cpp 文件中，那么需要在 EmployeeManagement.cpp 文件的开始部分写上以下语句：

```
include "EmployeeManagement.h"
```

8.1.4 类与结构体

结构体是 C++的一种重要的自定义数据类型，通过定义一个结构体，可以把多种类型的变量组合在一起，形成一个具有多个属性的新类型。C++的结构体是从 C 语言发展而来的。在 C 语言中，结构体中只能有数据成员，不能有成员函数。C++扩展了 C 语言结构体的功能，C++语言中的结构体不仅可以拥有数据成员，还可以拥有成员函数，可以通过函数成员直接操作结构体中的数据成员。这样把数据与操纵数据的函数综合在一起，即实现了“封装”。

在 C++中，使用 struct 和 class 都可以定义一个类。以下代码所定义的类与之前使用 class 定义的类是完全相同的。

```
struct Employee
{  private:
      string employeeid;
      string name;
      int age;
   public:
      void display( )
      {
         cout<<"编号:"<< employeeid<<endl;
         cout<<"姓名:"<< name <<endl;
         cout<<"年龄:"<< age <<endl;
      }
};
```

C++中使用 struct 和 class 进行类声明的主要区别为：在 struct 中，不使用 private、public 和 protected 限定符定义的数据成员和成员函数被编译器默认为是 public 的。而在 class 中没有使用限定符修饰的成员默认为 private。

```
struct Employee     //使用 struct 声明的类
{    //3 个数据成员为 public 成员
     string employeeid;
     string name;
     int age;
 public:
     void display( )
     {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
     }
};
class Employee      //使用 class 声明的类
```

```
{   //3 个 private 数据成员
     string employeeid;
     string name;
     int age;
 public:
     void display( )
     {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
     }
};
```

class 是 C++为了进行面向对象程序设计而引入的关键字，而 struct 是从 C 语言继承而来的，虽然两者都可以实现类的声明，但是使用 class 更能体现面向对象的特点。所以 C++中进行类的声明主要使用 class。

8.2　对象

8.2.1　对象的定义

C++中类是一种自定义数据类型，对象可以理解为使用“类”类型定义的变量。同结构体变量的定义方式相同，对象的定义方式也有三种：

（1）先声明类，再定义对象

如果使用 8.1.3 中所声明的 Employee 类进行对象的定义，语句为：

```
Employee  employee1, employee2;
```

表示使用 Employee 类定义了两个变量 employee1 和 employee2。这种定义方式与 C++标准类型变量定义是一样的。

（2）声明类的同时定义对象

```
class Employee
{  private:
     string employeeid;
     string name;
     int age;
   public:
     void display( )
     {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
     }
} employee1, employee2;
```

（3）直接定义对象

```
class
{  private:
```

```
    string employeeid;
    string name;
    int age;
    public:
    void display()
    {
       cout<<"编号:"<< employeeid<<endl;
       cout<<"姓名:"<< name <<endl;
       cout<<"年龄:"<< age <<endl;
    }
} employee1, employee2;
```

以上三种对象定义方式都是合法的，不过第三种方式一般很少使用。由于 C++程序经常把类的声明和对象的定义分别放在.h 文件和.cpp 文件中，所以 C++程序中对象的定义主要使用第一种方式，当程序规模较小时也可以采用第二种方式。

8.2.2 对象成员的引用

对象是类类型的变量，与结构体变量类似，对象在被使用时经常需要引用其成员。对象成员的引用包括数据成员的引用和成员函数的引用。需要注意的是，外部程序引用对象的成员时，只能引用对象的公有成员，而不能引用其私有和保护成员。

引用对象成员的一般形式为：

```
对象名.成员名
```

类作为自定义类型也可以定义指针变量，让指针变量指向一个具体的对象。当使用对象指针时，引用指针所指向的对象成员的形式可以为以下两种:

```
(*指向对象的指针).成员名
指向对象的指针->成员名
```

需要注意的是，使用指针访问对象时，必须首先执行指针的指向操作，使指针指向一个具体的对象，然后使用指针进行对象成员的引用。

【例 8.3】 引用对象成员。

```
#include <iostream>
using namespace std;
class Date
{ public:
    int month;
    int day;
    int year;
    void display()
    {
       cout<<year<<"-"<< month<<"-"<< year<<endl;
    }
};
int main()
{
```

```
    Date date1, * date2;
    date1.month=5;
    date1.day =20;
    date1.year=2010;
    date1.display( );
    date2=& date1;
    date2-> month =6;
    date2-> day =15;
    date2->year=2009;
    date2->display( );
    return 0;
}
```

程序执行结果为：

2010-5-20

2009-6-15

8.3　构造函数

8.3.1　构造函数的作用

使用类定义对象后，在引用对象成员时必须保证对象的数据成员已经被赋值，也就是说对象必须经过初始化才能被使用。在例 8.3 中，使用以下语句给 date1 对象的各个数据成员赋值。

```
date1.month=5;
date1.day =20;
date1.year=2010;
```

以上赋值过程也可以通过定义对象时进行初始化来完成。date1 的初始化可以如下进行:

```
Data date1={5, 20, 2010};
```

需要注意的是，无论是对数据成员赋值还是执行对象的初始化，都只能针对对象的公有数据成员来进行。如果对一个对象的私有数据成员赋值，会出现编译错误。假设 date 类的 year 数据成员改成 private，则以下操作将会被 C++编译器报错：

```
date1.year=2010;
```

在 VC++ 2005 中报告以下错误：

error C2248: "Date:: year": 无法访问 private 成员(在“Date”类中声明)

那么如何对一个对象的私有数据成员进行初始化呢？类内部的成员函数可以访问类中所有的数据成员。可以设计一个成员函数，使用该函数进行对象的初始化操作。

【例 8.4】 使用类的成员函数做对象的初始化操作。

```
#include <iostream>
using namespace std;
class Date
{ private:
    int month;
```

```
    int day;
    int year;
  public:
    void display()
    {
       cout<<year<<"-"<<month<<"-"<<day<<endl;
    }
    void setDate(int m,int d,int y)
    {
       month=m;
       day=d;
       year=y;
    }
};
int main()
{
    Date date1, date2;
    date1.setDate(5,20,2010);
    date2.setDate(6,15,2009);
    date1.display();
    date2.display();
    return 0;
}
```

程序执行结果为：

2010-5-20

2009-6-15

在例 8.4 中，类 Date 中定义了一个公有函数 setDate，使用该函数给对象的数据成员赋值，从而完成初始化操作。使用这类函数进行对象的初始化具有以下两个缺点：

（1）为了实现对象的初始化引入了初始化函数，增加了类的复杂性，由于该函数被声明为公有的，增加了一个由外部程序访问内部私有数据成员的接口。

（2）定义对象后如果忘记调用初始化函数或者对一个对象进行了多次初始化函数的调用，都会引起程序出错。

为了解决对象的初始化问题，C++提供了一个特殊的成员函数来完成对象的初始化操作，即**构造函数**(constructor)。构造函数具有以下几个特点：

（1）构造函数的目的就是实现对象的初始化；

（2）构造函数的名字与类名相同，不能由用户命名；

（3）构造函数没有函数返回值；

（4）构造函数不需要用户调用，对象建立时构造函数会自动执行。

【例 8.5】 使用类的构造函数实现对象的初始化。

```
#include <iostream>
using namespace std;
class Date
{ private:
    int month;
```

```
    int day;
    int year;
  public:
    Date( )  //缺省构造函数
    {
        month=0;
        day=0;
        year=0;
    }
    void display( )
    {
        cout<<year<<"-"<<month<<"-"<<day<<endl;
    }
    void setDate(int m,int d,int y)
    {
        month=m;
        day=d;
        year=y;
    }
};
int main( )
{
    Date date1, date2;
    date1.display( );
    date1.setDate(5,20,2010);
    date1.display( );
    date2.display( );
    return 0;
}
```

程序运行结果为：

```
0-0-0
2010-5-20
0-0-0
```

在例 8.5 中，类 Date 提供了一个构造函数来设置数据成员的默认值，该构造函数没有参数。C++中把类的无参构造函数称为**缺省构造函数。**构造函数不能像其他成员函数一样被显式调用。构造函数在对象被创建时由编译器隐式调用，在本例中在语句 Date date1, date2;执行后对象 date1 和 date2 会被创建，C++编译器会在创建对象时隐式地调用构造函数来完成对象的初始化。

8.3.2　带参数的构造函数

除了缺省构造函数，C++的类还允许使用带有参数的构造函数。在执行初始化操作时，可以把参数值赋给对象的数据成员。当 C++使用带参数的构造函数时，对象的定义形式必须和构造函数中参数的形式一致。

如带参数的构造函数的形式为 Date(int m, int d, int y)，则定义对象时必须按照以下形式进行：

Date date(5,15,2010);

对象名后边括号中的初值类型必须与构造函数对应的参数类型一致，否则会引发编译错误。

【例 8.6】 使用类的带参数的构造函数实现对象的初始化。

```
#include <iostream>
using namespace std;
class Date
{ private:
   int month;
   int day;
   int year;
  public:
   Date(int m,int d,int y) //带参数的构造函数
   {
       month=m;
       day=d;
       year=y;
   }
   void display()
   {
       cout<<year<<"-"<<month<<"-"<<day<<endl;
   }
};
int main()
{
   Date date1(5,20,2010), date2(6,15,2009);
   date1.display();
   date2.display();
   return 0;
}
```

程序运行结果为：

2010-5-20

2009-6-15

在例 8.6 中，类的构造函数有 3 个参数，当对象被定义时，构造函数被 C++编译器调用，此时对象的定义必须按照构造函数的格式进行。

8.3.3 构造函数重载

一个类可以拥有多个构造函数，即构造函数可以实现重载。当一个类拥有多个构造函数时，对象初始化操作可以使用其中任何一个来进行。哪个构造函数被调用取决于对象初始化时采用的形式。

【例 8.7】 构造函数的重载。

```
#include <iostream>
using namespace std;
class Date
{ private:
```

```
    int month;
    int day;
    int year;
  public:
    Date( )                              //缺省构造函数
    {
        month=0;
        day=0;
        year=0;
    }
    Date(int m,int d,int y)      //带参数的构造函数
    {
        month=m;
        day=d;
        year=y;
    }
    void display( )
    {
        cout<<year<<"-"<<month<<"-"<<day<<endl;
    }
};
int main( )
{
    Date date1(5,20,2010);       //调用构造函数 Date(int m,int d,int y)
    Date date2;                  //调用构造函数 Date( )
    date1.display( );
    date2.display( );
    return 0;
}
```

程序运行结果为：

```
2010-5-20
0-0-0
```

使用构造函数需要注意以下两点：

(1) C++的每次对象创建都会伴随着构造函数的调用，即使用户没有给类定义构造函数。在例 8.4 中并没有给 Date 类设计构造函数，在执行 Date date1, date2;对象定义语句时，C++编译器会给该类提供一个空的缺省构造函数。该构造函数虽然被调用，但是什么都不做。

(2) 只要用户在类中定义了构造函数，C++编译系统将不会再提供缺省构造函数。此时定义对象只能按照构造函数中参数的个数和类型进行。如例 8.6 中声明的 Date 类只有一个构造函数 Date(int m,int d,int y)，该构造函数提供了对象的定义规范。如下两个对象的定义语句：

```
Date birthday(10,15,1988);  //正确，对象的定义与类的构造函数形式一致
Date birthday;              //错误，对象的定义与类的构造函数形式不一致
```

所以在类的声明中，可以通过增加一个缺省构造函数来消除以上对象定义时形式不一致引发的错误。

8.3.4 拷贝构造函数

类是 C++中的自定义类型，对象是一个类类型的变量。当声明了一个类 Date 后，可以使用以下语句进行对象定义：

```
Date birthday;
```

对于 C++的标准数据类型，可以在定义一个变量的同时使用另一变量对其进行初始化，例如：

```
int x = 10;
int y = x;
```

作为 C++自定义类型的类，在定义对象时也可以使用基于同一个类的其他对象对其进行初始化。如使用例 8.6 中所定义的 Date 类，可以有如下定义：

```
Date date1(5,20,2010);
Date date2=date1;
```

在上述两个类的定义语句中，第一条语句调用类 Date 的构造函数 Date(int m,int d,int y)进行对象的初始化；而第二条语句中对 date2 进行初始化时调用了类 Date 的一个特殊构造函数——**拷贝构造函数**(copy constructor)。拷贝构造函数用来创建一个新的对象，该对象是另外一个对象的复制品。对于 Date 类，可以设计如下形式的拷贝构造函数:

```
Date::Date(const Date& d)
{
     month=d.month;
     day=d.day;
     year=d.year;
}
```

上述拷贝构造函数只有一个参数，该参数是 Date 对象的引用形式，前边加上 const 声明，可以保证函数调用过程中函数体内不会修改该对象的值。当执行对象复制时，拷贝构造函数被调用，它把被复制对象的各个数据成员的值分别赋给新对象的对应数据成员，从而完成对象的拷贝工作。

如果类没有定义拷贝构造函数，C++会提供默认的拷贝构造函数，该拷贝构造函数的功能是把被复制对象的数据成员一一对应地复制到新对象中。因此对 Date 类来说，系统提供的默认拷贝构造函数与我们设计的拷贝构造函数的功能是一样的。

【例 8.8】 使用拷贝构造函数。

```
#include <iostream>
using namespace std;
class Date
{ private:
   int month;
   int day;
   int year;
  public:
  Date(int m,int d,int y) //带参数的构造函数
```

```
    {
      month=m;
      day=d;
      year=y;
    }
    Date(const Date& d)        //拷贝构造函数
    {
      month=d.month;
      day=d.day;
      year=d.year;
      cout<<"拷贝构造函数被调用! "<<endl;
    }
    void display( )
    {
      cout<<year<<"-"<<month<<"-"<<day<<endl;
    }
};
int main( )
{
    Date date1(5,20,2010);  //调用构造函数 Date(int m,int d,int y)
    Date date2=date1;       //调用拷贝构造函数 Date(const Date& d)
    date1.display( );
    date2.display( );
    return 0;
}
```

程序运行结果为:

拷贝构造函数被调用!
2010-5-20
2010-5-20

8.4　析构函数

当一个对象被撤销时，撤销前会调用类的析构函数。**析构函数**(destructor)是类的一个特殊成员函数，析构函数的函数名是在类名的前边加上一个“～”符号。

与构造函数一样，析构函数没有函数返回值；但是与构造函数不同，析构函数没有函数参数，所以析构函数不能被重载。一个类只能拥有一个析构函数。

如果用户没有给类提供析构函数，那么 C++编译器会自动地提供一个默认析构函数，该析构函数不执行任何操作。

如果对象使用了动态内存或其他资源，用户就必须为该类提供析构函数，在析构函数中完成内存或其他资源的回收工作以防止内存泄漏。

【例 8.9】 使用构造函数和析构函数。

```
#include <iostream>
#include <string>
using namespace std;
```

```
class Date
{ private:
   int month;
   int day;
   int year;
  public:
   Date(int m,int d,int y) //带参数的构造函数
   {
        month=m;
        day=d;
        year=y;
   }
   ~Date()                 //析构函数
   {
        cout<<year<<"-"<<month<<"-"<<day <<"析构函数被调用! "<<endl;
        }
        void display()
        {
        cout<<year<<"-"<<month<<"-"<<day<<endl;
        }
};
int main()
{
   Date date1(5,20,2010);
   Date date2(10,15,2009);
   date1.display();
   date2.display();
   return 0;
}
```

程序运行结果为：

2010-5-20

2009-10-15

2009-10-15 析构函数被调用!

2010-5-20 析构函数被调用!

Date 类中定义了析构函数，由于对象没有执行动态申请内存或其他资源的操作，该析构函数不用释放资源。在析构函数体内写了一条输出语句，可以通过程序是否执行了输出语句来判断析构函数是否被调用。对象 date1 在 main 函数中定义为全局对象，当 main 函数运行结束后全局对象才会被撤销。当对象被撤销前析构函数被调用，从运行结果可以看出，首先被构造的对象最后被析构，即构造的顺序与析构的顺序相反。

8.5 类的静态成员

到目前为止，使用前边声明的类定义对象时，每个对象都拥有各自的数据成员和成员函数。使用例 8.4 中声明的 Date 类定义两个对象 date1、date2，例如：

```
Date date1(5,20,2010);
Date date2(6,15,2009);
```

则 date1 和 date2 分别拥有各自的 month、day 和 year 数据成员，并进行了数据成员的初始化。

C++提供了另外一种成员，这种成员只属于类本身，不属于类的对象，这种成员称做**类的静态成员**。类的静态成员包括静态数据成员和静态成员函数。

8.5.1　静态数据成员

如果给类的某个数据成员的声明语句前边添加 static 关键字，则该成员就成为类的静态数据成员。例如：

```
class Date
{
   private:
     int month;
     int day;
     int year;
   public:
     static int n;  //类的静态数据成员
};
```

静态数据成员不属于任何一个对象，静态数据成员只占有一份内存，每个对象都可以引用这个静态数据成员，该数据成员的值是被所有该类的对象所共享的，如果修改了该静态数据成员的值，则该类所有对象中该静态数据成员的值都被修改了。

静态数据成员可以被初始化，不过初始化语句只能放在类外。静态数据成员的初始化形式为：

```
数据类型 类名::静态数据成员名=初值;
```

例如：`int Date::n=0;`

【例 8.10】 使用类的静态数据成员。

```
#include <iostream>
#include <string>
using namespace std;
class Date
{ private:
    int month;
    int day;
    int year;
  public:
     static int n;
  Date(int m,int d,int y)   //带参数的构造函数
  {
    month=m;
    day=d;
    year=y;
    n++;
  }
  Date(const Date& d)       //拷贝构造函数
```

```
    {
      month=d.month;
      day=d.day;
      year=d.year;
      n++;
    }
    ~Date()                    //析构函数
    {
      n--;
    }
    void display()
    {
      cout<<year<<"-"<<month<<"-"<<day<<endl;
  }
  };
  int Date::n=0;
  int main()
  {
     Date date1(5,20,2010);
     cout<<"Date 对象的个数为:"<< Date::n <<endl;
     cout<<"Date 对象的个数为:"<<date1.n <<endl;
     Date date2=date1;
     cout<<"Date 对象的个数为:"<< Date::n <<endl;
     cout<<"Date 对象的个数为:"<<date2.n <<endl;
     date1.display();
     date2.display();
     return 0;
  }
```

程序运行结果为：

Date 对象的个数为：1
Date 对象的个数为：1
Date 对象的个数为：2
Date 对象的个数为：2
2010-5-20
2010-5-20

例 8.10 在类 Date 中定义了一个静态数据成员 n，用它来表示当前类 Date 类型的对象的个数，即 n 起到对象计数器的作用，用来统计当前程序中对象的个数。在类的构造函数和拷贝构造函数中，让静态数据成员 n 执行加 1 操作，在程序中当类 Date 的一个对象被创建时对象的个数加 1，其构造函数或拷贝构造函数被调用，n 随之加 1。在类的析构函数中让静态数据成员 n 执行减 1 操作，当类 Date 的一个对象被撤销时对象的个数减 1，其析构函数被调用，n 随之减 1。

8.5.2 静态成员函数

与静态数据成员的定义方式一样，在类中成员函数的声明前加上 static，该成员函数就成为类的静态成员函数。

```
class Date
{
   private:
     int month;
     int day;
     int year;
   public:
     static int n;  //类的静态数据成员
     static void displayObjectNum( )
     {
         cout<<"Date 对象的个数为:"<<n<<endl;
     }
};
```

类的静态成员函数只能访问类的静态成员(包括静态数据成员和静态成员函数)，不能访问类内的其他数据成员和成员函数。

【例 8.11】 使用类的静态数据成员和静态成员函数。

```
#include <iostream>
#include <string>
using namespace std;
class Date
{ private:
   int month;
   int day;
   int year;
   static int n;
  public:
     Date(int m,int d,int y)     //带参数的构造函数
     {
        month=m;
        day=d;
        year=y;
        n++;
     }
     Date(const Date& d)         //拷贝构造函数
     {
        month=d.month;
        day=d.day;
        year=d.year;
        n++;
     }
     ~Date( )                    //析构函数
     {
        n--;
     }
     void display( )
     {
```

```
        cout<<year<<"-"<<month<<"-"<<day<<endl;
    }
    static void displayObjectNum()
    {
        cout<<"Date 对象的个数为:"<<n<<endl;
    }
};
int Date::n=0;
int main()
{
     Date date1(5,20,2010);
     Date::displayObjectNum();
     Date date2=date1;
     Date::displayObjectNum();
     date1.display();
     date2.display();
     return 0;
}
```

程序运行结果为：

Date 对象的个数为：1

Date 对象的个数为：2

2010-5-20

2010-5-20

例 8.11 把静态数据成员 n 声明为私有的，这样可以防止外部程序修改 n 的值。由静态成员函数 displayObjectNum 来输出对象的个数。

8.6 友元

类的数据成员和成员函数可以分为公有(public)、私有(private)，公有的成员可以被类外的程序代码访问，私有成员只能被本类内的成员函数访问。

有些情况下需要开放类的私有成员给特定的函数或类来访问，这时可以把该特定函数或类声明为一个类的友元(friend)，友元可以访问具有友好关系的类的私有成员。友元包括友元函数和友元类。

8.6.1 友元函数

如果一个函数想成为一个类的友元函数，则必须在该类中声明该函数为友元函数。声明的方法：在类中写上函数的声明，并在前边加上 friend 关键字。当声明一个类的友元函数后，就可以使用该函数访问类中的私有成员。类的友元函数可以是另一个类的成员函数，也可以是不属于任何一个类的普通函数。

【例 8.12】 使用友元函数访问类的私有成员。

```
#include <iostream>
#include <string>
```

```
using namespace std;
class Date
{ private:
   int month;
   int day;
   int year;
  public:
   friend void modifyDate(Date& date,int month,int day,int year);
                                        //声明类 Date 的友元函数
    Date(int m,int d,int y)     //带参数的构造函数
    {
        month=m;
        day=d;
        year=y;
    }
    Date(const Date& d)              //拷贝构造函数
    {
        month=d.month;
        day=d.day;
        year=d.year;
    }
    void display( )
    {
        cout<<year<<"-"<<month<<"-"<<day<<endl;
    }
};
void modifyDate(Date& date,int month,int day,int year)  //友元函数的定义
{
    date.month=month;
    date.day=day;
    date.year=year;
}
int main( )
{
    Date date1(5,20,2010);
    Date date2=date1;
    date1.display( );
    date2.display( );
    modifyDate(date1,6,15,2009);
    modifyDate(date2,8,8,2008);
    date1.display( );
    date2.display( );
    return 0;
}
```

程序运行结果为：

```
2010-5-20
2010-5-20
```

2009-6-15
2008-8-8

在例 8.12 中，类 Date 的数据成员都是私有的，对象通过构造函数来初始化各个数据成员的值，通过 display 成员函数来显示各个数据成员的值。但是类 Date 的对象不能修改初始化后的各个数据成员的值。这里通过类 Date 的友元函数 modifyDate 来修改私有数据成员。把 modifyDate 函数的声明写在类 Date 内，加上关键字 friend 指示其为类 Date 的友元函数。在类外实现函数的定义，函数接收一个对象的引用和三个整型参数，在函数体内修改该对象的各个数据成员。

8.6.2 友元类

一个函数可以是一个类的友元，一个类也可以是另一个类的友元。假设有类 A 和类 B，如果声明 A 是 B 的友元类，则 A 中所有的函数都自动成为 B 的友元函数，可以通过 A 的成员函数来访问 B 类中的所有成员。

在 B 类中使用如下语句来声明 A 为其友元类：

```
friend A;
```

【例 8.13】 使用友元类。

```
#include <iostream>
#include <string>
using namespace std;
class DateFriend;       //类声明
class Date
{
private:
friend DateFriend;      //定义友元类
int month;
int day;
int year;
public:
Date(int m,int d,int y)
{
    month=m;
    day=d;
    year=y;
}
};
class DateFriend
{
public:
void modifyDate(Date& date,int month,int day,int year)
{
    date.month=month;
    date.day=day;
    date.year=year;
}
```

```
    void display(const Date& date)
    {
        cout<<date.year<<"-"<<date.month<<"-"<<date.day<<endl;
    }
    };
    int main( )
    {
        Date date1(5,20,2010);
        Date date2(6,15,2009);
        DateFriend DateFriend1;
        DateFriend1.modifyDate(date1,8,12,2008);
        DateFriend1.display(date1);
        DateFriend1.display(date2);
        return 0;
    }
```

程序运行结果为：

2008-8-12

2009-6-15

例 8.13 中类 Date 只有一个构造函数，没有提供数据成员的修改功能。为了能在外部修改其私有数据成员，程序中定义了一个友元类来完成类 Date 私有成员的修改和显示。在类 Date 中使用 friend 关键字声明一个友元类 DateFriend，在 DateFriend 中通过 modifyDate 函数修改 Date 对象的私有数据成员，通过 display 函数显示 Date 对象的私有数据成员。

8.7　VC++ 2005 中使用类向导

在 VC++ 2005 中创建 Win32 控制台应用程序项目后，可以在新添加的 cpp 文件中手工输入代码完成类的声明，也可以借助类向导来完成类的声明过程。添加一个类后，还可以通过向导添加数据成员和成员函数。使用类向导可以快速准确地声明一个类及添加相关成员。打开“添加类”向导的方法有两种：一是通过菜单“项目”→“添加类”，二是在解决方案资源管理器中，右键单击项目文件名，选择右键菜单“添加”→“添加类”。结果如图8-3所示。

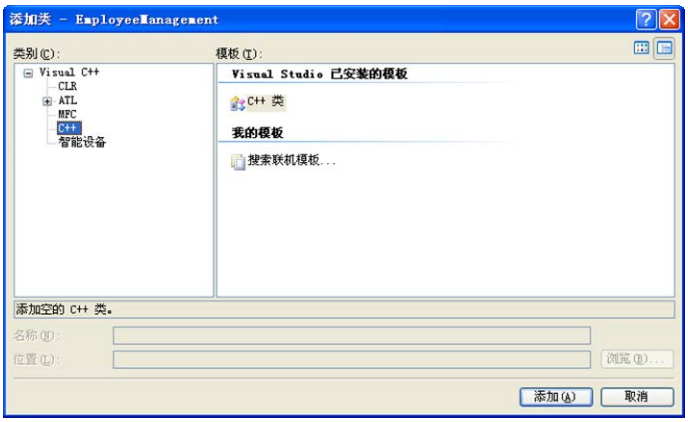


图 8-3　“添加类”向导

在向导左边的树状目录中选择 C++，在显示的界面中点击 C++类，然后点击“添加”按钮。将出现“一般 C++类向导”，如图8-4所示。在图8-4中“类名”下边的文本框中输入需要声明的类的名称。

图 8-4　一般 C++类向导

如果想把当前声明的类放入以前的文件中，可以先删掉这两个文本框中的文件名，然后单击“.h 文件”和“.cpp 文件”文本框右边的按钮来选择需要加入的文件，如图8-5所示，最后单击“完成”按钮即可。向导中第二行文本框是做类的继承和继承访问控制的，如果当前声明的类没有父类，可以省略基类名称。

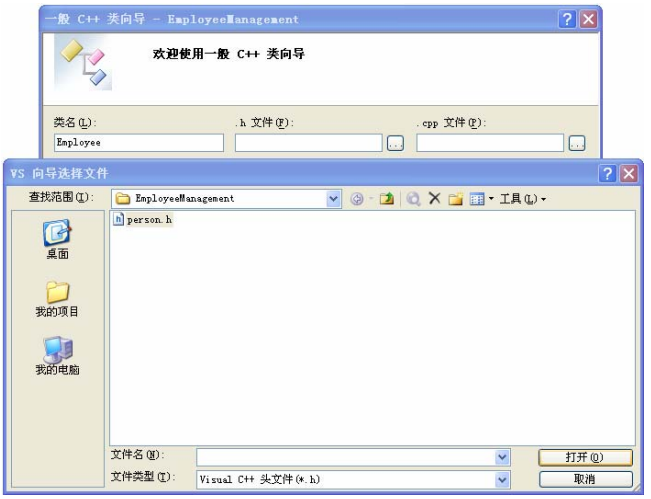


图 8-5　把类的声明存入其他文件

假设如图 8-4 所示，在一个空项目中声明了 Employee 类，其声明和实现分别存放在 Employee.h 和 Employee.cpp 文件中，打开 Employee.h 文件，将看到如下代码：

```
#pragma once
class Employee
{
```

```
    public:
      Employee(void);
    public:
      ~Employee(void);
};
```

第一行代码#pragma once 为编译器指令，用来指示无论被多少个.cpp 文件 include，程序被编译时该.h 文件只被编译一次。后边的代码即为类 Employee 的声明，类中声明了一个构造函数和一个析构函数。在 Employe.cpp 文件中自动生成了如下代码：

```
#include "Employee.h"
Employee::Employee(void)
{
}
Employee::~Employee(void)
{
}
```

点击菜单“视图”→“类视图”，可以打开在当前程序中所声明的类及其结构视图，如图 8-6 所示。

在图8-6 中 Employee 类上点击右键，在弹出的菜单中（如图8-7 所示）选择添加变量可以给 Employee 类增加数据成员。

图 8-6　类视图

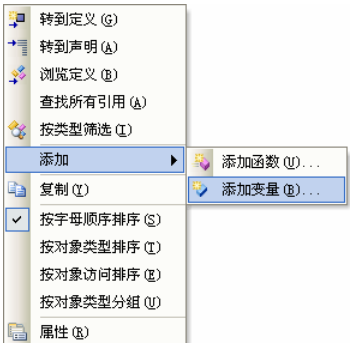


图 8-7　添加新成员

在图8-8 中给 Employe 类添加一个公有的 Employeeid 数据成员，该成员设为字符串。需要注意的是，由于 string 是 C++标准库提供的一个类，不是 C++的标准数据类型，所以在变量类型中不能通过下拉菜单选择，而是人工输入 string。点击“完成”按钮结束类数据成员的添加过程。

给类添加成员函数的过程与添加数据成员类似，在图8-7 菜单中选择“添加函数”，在图8-9 的界面中可以设置成员函数的各个部分，包括返回类型、函数名、参数类型、参数列表、访问控制方法等。点击“完成”按钮结束成员函数的添加过程。

图 8-8　添加成员变量向导

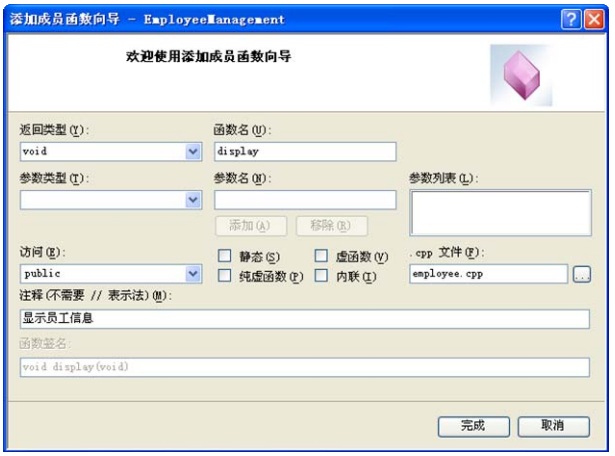


图 8-9　添加成员函数向导

本 章 小 结

类和对象是面向对象程序设计中的重要概念。类是事物的抽象，对象是类的实例。在 C++ 语言中类也是一种用户自定义类型，类中包括数据成员和成员函数，对象则是“类”类型的变量。构造函数和析构函数是类的特殊成员函数，它们分别用于对象创建时的初始化和对象撤销前的资源清理操作。可以通过设置静态成员实现对象间的数据共享。通过设计友元来实现对类私有成员的外部存取。友元破坏了类的封装特性，在使用过程中需要慎重选择。

习　题　8

一、选择题

1. 假定 Student 为一个类，则该类的拷贝构造函数的声明语句为(　　)。

A. student&(Student x);　　B. student(Student x);

C. student(Student &x);　　D. student (Student* x)

2. 对于使用 struct 定义的类，若其成员没有访问控制修饰符指明访问控制权限，则其隐含访问权限为(　)。

A. public　　B. private　　C. protected　　D. static

3. 假定 student 为一个类，语句 student stu1，*p；会自动调用类 Student 的构造函数的次数为(　)。

A. 1　　B. 2　　C. 3　　D. 4

4. 以下类的定义中，横线处应该填入的内容为(　)。

```
    class student
{
    public:
      string name;
      int age;
      static int classno;
};
_______ classno=3;
```

A. int　　B. int Student::　　C. static int　　D. static int Student::

5. 关于友元的说法正确的是(　)。

A. 类的友元函数不能访问类的私有成员

B. 类的友元函数属于该类

C. 类 a 为类 b 的友元类，则 a 的所有成员函数均可访问类 b 的任何成员

D. 类的友元加强了类的封装性

二、分析程序的运行结果，并加以解释

(1)

```
#include<iostream >
using namespace std;
    class node
{
    int x,y;
    public:
        node() {x=y=0;}
    node(int a,int b) {x=a;y=b;}
    ~node()
    {
        cout<<"node("<<x<<","<<y<<")被撤销"<<endl;
    }
    void disp()
    {
        cout<<"x="<<x<<",y="<<y<<endl;
    }
};
void main()
{
    node n1, n2(10,15);
```

```
    n1.disp();
    n2.disp();
}
```

(2)

```
#include<iostream>
using namespace std;
class node
{
   int x;
   static int y;
   public:
      node(int a){x=a,y=2*a;}
      static void disp(node n);
};
void node::disp(node n)
{
  cout<<" node( "<<n.x<<","<<y<<")"<<endl;
}
int node::y=0;
void main()
{
   node n1(3),n2(5);
   node::disp(n1);
   node::disp(n2);
}
```

三、简答题

1. 类和对象的关系是什么?
2. 构造函数和析构函数各自的功能是什么？都是在什么时候被调用?
3. 使用 class 定义的类和使用 struct 定义的类有什么区别?
4. 友元函数的作用是什么？在什么情况下需要使用友元函数?

四、编程题

设计一个长方体类 Box，用户从键盘输入长、宽、高，计算该长方体的体积，要求提供两个构造函数，在缺省构造函数中设置长、宽、高均为 0，带有 3 个参数的构造函数能够使用用户的输入值进行初始化。

第 9 章　类的继承、派生与多态

9.1　类的继承与派生

9.1.1　继承与派生的概念

C++中可以通过一个已有的类派生一个新类，这种机制称为类的继承。继承是 C++的一个重要组成部分，是面向对象程序设计的重要特征。通过继承可以重用已有的程序代码和设计，从而提高程序开发效率。

以下是在第 8 章中员工类的声明：

```
class Employee
{  private:
    string employeeid;
    string name;
    int age;
  public:
    void display()
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
    }
    void setEmployee(string inputid,string inputname,int inputage)
    {
        employeeid=inputid;
        name=inputname;
        age=inputage;
    }
};
```

该员工类具有员工号、姓名、年龄三个数据成员和两个成员函数：一个用来显示数据成员的成员函数display，一个用来输入数据成员的setEmployee。假设企业中员工岗位类型分为技术型和管理型，现在程序中需要设计一个新的类，用来描述技术型员工。由于“技术型员工”是“员工”的一种，也具有员工号、姓名、年龄这些基本属性，所以可以从“员工”类派生出“技术型员工”类，此时称“技术型员工”类为“员工”类的**派生类**，“员工”类为“技术型员工”类的**基类**。也可以把“技术型员工”类称为“员工”类的**子类**，“员工”类为“技术型员工”类的**父类。**

为了使派生类“技术型员工”中能够拥有基类“员工”中的数据成员，首先修改一下“员工”类的声明，然后再做类的继承。

```
class Employee
{  protected:            //修改私有成员为保护成员
    string employeeid;
    string name;
    int age;
  public:
    void display()
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
    }
    void setEmployee(sting inputid,string inputname,int inputage)
    {
        employeeid= inputid;
        name= inputname;
        age=inputage;
    }
};
```

以下为“技术型员工”类的声明：

```
class Technican: public Employee
{  private:
    string level;           //增加一个数据成员，表示岗位级别
  public:
    void displaylevel()     //增加一个显示岗位级别的成员函数
    {
        cout<<"岗位级别:"<<level<<endl;
    }
};
```

现在 Technican 类首先具有了 Employee 类的三个保护的数据成员和公有的成员函数，然后增加了一个新的私有数据成员和一个公有的成员函数。图9-1 表示类 Employee 和 Technican 的继承关系。箭头从派生类指向基类。

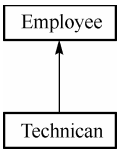


图 9-1　类 Employee 和 Technican 的继承关系

9.1.2　派生类定义的格式

声明派生类的一般形式为：

```
class 派生类名:[继承方式] 基类名
   {
     派生类新增加的成员
   };
```

继承方式包括：public(公有)，private(私有)和 protected(保护)。继承方式为可选项，如果在声明派生类时没有显式地指出继承方式，则默认为 private(私有)。

如果派生类中新增加的成员有与基类同名的数据成员，则派生类的成员会覆盖掉基类的同名成员，在派生类中访问该成员时访问的是派生类的成员，而不是基类中的同名成员。

如果派生类中新增加的成员有与基类同名的成员函数，并且参数的个数和类型也相同，派生类的成员函数会覆盖掉基类的同名成员函数。如果只是名字相同而参数不同，该成员函数会和基类中的同名成员函数构成函数重载。

【例 9.1】 使用派生类。

```
#include<iostream>
#include<string>
using namespace std;
class Employee
{  protected:
    string employeeid;
    string name;
    int age;
   public:
    void display()
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
    }
    void setEmployee(string inputid,string inputname,int inputage)
    {
        employeeid= inputid;
        name= inputname;
        age=inputage;
    }
};
class Technican: public Employee
{ private:
    string level;   //派生类的数据成员
  public:
    void display()  //派生类的成员函数
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
        cout<<"岗位级别:"<<level<<endl;
    }
    void setTechnican(string inputlevel)
    {
        level=inputlevel;
    }
};
int main()
{
    Employee employee1;
    Technican technican1;
    employee1.setEmployee("20100001","王军",25);
```

```
    employee1.display();
    technican1.setEmployee("20100002","李明",26);
    technican1.setTechnican("助理工程师");
    technican1.display();
    return 0;
}
```

程序运行结果为：

编号：20100001
姓名：王军
年龄：25
编号：20100002
姓名：李明
年龄：26
岗位级别：助理工程师

在例 9.1 中，派生类 Technican 和基类 Employee 具有相同的成员函数 display，在派生类中该成员函数实现了对基类同名函数的覆盖。

```
technican1.display();
```

语句会调用派生类的 display 成员函数。

例 9.1 中声明的 Employee 和它的派生类 Technican 都是没有构造函数的，它们提供公有函数来实现内部数据成员的初始化。当基类和派生类都使用构造函数进行数据成员的初始化时，需要注意以下两点：

(1) 构造函数是特殊的成员函数，构造函数不能被派生类继承。

(2) 当需要进行数据成员初始化时，派生类的构造函数不仅要初始化自己新增加的数据成员，还要注意对基类数据成员执行初始化操作。

派生类构造函数的一般形式为：

```
派生类构造函数名(总参数表列):基类构造函数名(参数表列)
        {派生类中新增的数据成员初始化语句}
```

当派生类的构造函数被调用时，首先执行基类中的构造函数对基类数据成员进行初始化，然后再初始化派生类中新增加的数据成员。

【例 9.2】 派生类的构造函数。

```
#include<iostream>
#include<string>
using namespace std;
class Employee
{
   protected:
   string employeeid;
   string name;
   int age;
public:
void display()
{
```

```
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
    }
    employee(string inputid,string inputname,int inputage)
    {
       employeeid= inputid;
       name= inputname;
       age=inputage;
    }
    };
    class technican: public employee
    { private:
        string level;   //派生类的数据成员
      public:
        void display()  //派生类的成员函数
        {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
        cout<<"岗位级别:"<<level<<endl;
        }
        //派生类的构造函数
        technican(string inputid,string inputname,int inputage,string inputlevel):
        employee(inputid,inputname,inputage)
        {
            level=inputlevel;
        }
    };
    int main()
    {
        employee employee1("20100001","王军",25);;
        technican technican1("20100002","李明",26,"助理工程师");
        employee1.display();
        technican1.display();
        return 0;
    }
```

程序运行结果为：

编号：20100001
姓名：王军
年龄：25
编号：20100002
姓名：李明
年龄：26
岗位级别：助理工程师

与构造函数类似，析构函数也不能被派生类继承，派生类中的析构函数用来清理派生类

所申请的资源，基类的清理工作仍然由基类的析构函数负责。当执行派生类的析构函数时，系统会自动调用基类的析构函数和派生类的析造函数。析构函数的调用顺序与构造函数相反，先执行派生类自己的析构函数，然后再调用基类的析构函数。

9.1.3 继承方式

在声明派生类后，派生类就继承了基类的数据成员和成员函数，但是这些成员并不都能直接被派生类所访问。采用不同的继承方式决定了基类成员在派生类中的访问属性。在派生类中，对基类的继承方式包括 public（公有）、private（私有）和 protected（保护）三种。

三种不同继承方式的特点如下。

（1）公有继承（public inheritance）

基类的公有成员和保护成员在派生类中保持原有访问属性，其私有成员仍为基类私有。

采用公有继承声明的派生类可以访问基类中的公有成员和保护成员，而基类的私有成员则不能被访问，如表 9-1 所示。

表 9-1 公有继承中派生类对基类的访问属性

基类成员	基类成员在派生类中的访问属性
公有成员	公有
保护成员	保护
私有成员	不能被访问

【例 9.3】 使用公有继承方式。

```
#include<iostream>
#include<string>
using namespace std;
class employee
{
  public:
    string employeeid;
    string name;
  private:
    double wage;
  protected:
    int age;
  public:
    void display()
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
        cout<<"工资:"<<wage<<endl;
    }
    void setemployee(string inputid,string inputname,int inputage,double
    inputwage)
    {
        employeeid= inputid;
```

```
        name= inputname;
        age=inputage;
        wage=inputwage;
    }
};
class eechnican: public employee
{
  private:
    string level;   //派生类的数据成员
  public:
    void display()  //派生类的成员函数
    {
       cout<<"编号:"<< employeeid<<endl; //正确! 在派生类内访问基类的公有成员
       cout<<"姓名:"<< name <<endl;      //正确! 在派生类内访问基类的公有成员
       cout<<"年龄:"<< age <<endl;       //正确! 在派生类内访问基类的保护成员
       cout<<"工资:"<<wage<<endl;        //错误! 在派生类内访问基类的私有成员
       cout<<"岗位级别:"<<level<<endl;
    }
    void settechnican(string inputlevel)
    {
        level=inputlevel;
    }
};
int main()
{
    technican technican1;
    technican1.setemployee("20100002","李明",26,2500);
    technican1.settechnican("助理工程师");
    technican1.display();
    return 0;
}
```

(2) 私有继承(private inheritance)

基类的公有成员和保护成员在派生类中成了私有成员，其私有成员仍为基类私有，如表 9-2 所示。

表 9-2　私有继承中派生类对基类的访问属性

基类成员	基类成员在派生类中的访问属性
公有成员	私有
保护成员	私有
私有成员	不能被访问

【例 9.4】 使用私有继承方式。

```
#include<iostream>
#include<string>
using namespace std;
class Employee
{
```

```
  public:
    string employeeid;
    string name;
  private:
    double wage;
  protected:
    int age;
  public:
    void display()
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
        cout<<"工资:"<<wage<<endl;
    }
    void setEmployee(string inputid,string inputname,int inputage,double
    inputwage)
    {
        employeeid= inputid;
        name= inputname;
        age=inputage;
        wage=inputwage;
    }
};
class Technican: private Employee      //私有继承方式
{
  private:
    string level;                      //派生类的数据成员
  public:
    void display()                     //派生类的成员函数
    {
    cout<<"编号:"<< employeeid<<endl;  //正确! 在派生类内访问基类的公有成员
    cout<<"姓名:"<< name <<endl;       //正确! 在派生类内访问基类的公有成员
    cout<<"年龄:"<< age <<endl;        //正确! 在派生类内访问基类的保护成员
    cout<<"工资:"<<wage<<endl;         //错误! 在派生类内访问基类的私有成员
    cout<<"岗位级别:"<<level<<endl;
    }
    void setTechnican(string inputlevel)
    {       level=inputlevel;   }
};
int main()
{   Technican technican1;
    //错误! setEmployee 现在为 Technican 类的私有函数，不能在类外被访问
    technican1.setEmployee("20100002","李明",26,2500);
    technican1.setTechnican("助理工程师");
    technican1.display();
    return 0;
}
```

在例9.4中，由于采用私有继承，基类中的公有函数成了派生类中的私有函数，所以不能使用以下语句进行对象的初始化。

```
technican1.setEmployee("20100002","李明", 26, 2500);
```

(3) 保护继承(protected inheritance)

基类的公有成员和保护成员在派生类中成了保护成员，其私有成员仍为基类私有，如表 9-3 所示。

表 9-3　保护继承中派生类对基类的访问属性

基 类 成 员	基类成员在派生类中的访问属性
公有成员	保护
保护成员	保护
私有成员	不能被访问

【例 9.5】 使用保护继承方式。

```
#include<iostream>
#include<string>
using namespace std;
class employee
{
  public:
    string employeeid;
    string name;
  private:
    double wage;
  protected:
    int age;
  public:
     void display()
     {
         cout<<"编号:"<< employeeid<<endl;
         cout<<"姓名:"<< name <<endl;
         cout<<"年龄:"<< age <<endl;
         cout<<"工资:"<<wage<<endl;
     }
     void setemployee(string inputid,string inputname,int inputage,double
     inputwage)
     {
         employeeid= inputid;
         name= inputname;
         age=inputage;
         wage=inputwage;
     }
};
class technican: protected Employee     //保护继承方式
{
  private:
    string level;                       //派生类的数据成员
```

```
  public:
    void display()                          //派生类的成员函数
    {
      cout<<"编号:"<< employeeid<<endl; //正确! 在派生类内访问基类的公有成员
      cout<<"姓名:"<< name <<endl;      //正确! 在派生类内访问基类的公有成员
      cout<<"年龄:"<< age <<endl;       //正确! 在派生类内访问基类的保护成员
      cout<<"工资:"<<wage<<endl;        //错误! 在派生类内访问基类的私有成员
      cout<<"岗位级别:"<<level<<endl;
    }
    void settechnican(string inputlevel)
    {   level=inputlevel;   }
};
int main()
{
    technican technican1;
    //错误! setemployee 现在为 technican 类的保护函数，不能在类外被访问
    technican1.setemployee("20100002","李明",26,2500);
    technican1.settechnican("助理工程师");
    technican1.display();
    return 0;
}
```

在例 9.5 中，由于采用保护继承，基类中的公有函数成了派生类中的私有函数，所以不能使用以下语句进行对象的初始化。

```
technican1.setemployee("20100002","李明",26,2500);
```

从例 9.3～例 9.5，不同的继承方式会使基类的成员(包括数据成员和成员函数)在派生类中具有不同的访问控制属性。

对于公有继承，基类的公有和保护成员在派生类中仍为公有和保护成员，所以在派生类中使用这些成员仍然可以按照基类中的访问控制方式进行。

对于私有继承，基类的公有和保护成员在派生类转变为私有成员，这些成员只能在派生类内被访问，不能提供给外部程序使用。

对于保护继承，基类的公有和保护成员在派生类转变为保护成员，与私有继承一样，这些成员也只能在派生类内被访问。

无论是公有继承、私有继承，还是保护继承，基类中的私有成员在派生类中都不可访问的，只能被基类自身的成员所访问。如果想在派生类中访问基类的私有数据成员，只有通过使用基类的公有成员函数间接地来访问。

【例 9.6】 在派生类中使用基类的公有成员函数访问基类私有成员。

```
#include<iostream>
#include<string>
using namespace std;
class employee
{
  public:
    string employeeid;
    string name;
```

```
  private:
    double wage;
  protected:
    int age;
  public:
    void display()
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
        cout<<"工资:"<<wage<<endl;
    }
    void setemployee(string inputid,string inputname,int inputage,double
     inputwage)
    {
        employeeid= inputid;
        name= inputname;
        age=inputage;
        wage=inputwage;
    }
    };
    class technican: public employee        //公有继承
    {
    private:
        string level;                       //派生类的数据成员
    public:
        void display ()                     //派生类的成员函数
        {
            employee::display();            //调用基类的公有成员函数
            cout<<"岗位级别:"<<level<<endl;
        }
        void settechnican(string inputid,string inputname,int inputage,
        double inputwage,string inputlevel)
        {
            setemployee(inputid,inputname,inputage,inputwage);
                                            //调用基类的公有成员函数
            level=inputlevel;
        }
};
int main()
{
    technican technican1;
    technican1.settechnican("20100002","李明",26,2500,"助理工程师");
    technican1.displaytechnican();
    return 0;
}
```

程序运行结果为：

编号：20100002
姓名：李明
年龄：26
工资：2500
岗位级别：助理工程师

在例 9.6 中，在派生类 Technican 中，通过使用基类的公有成员函数 setemployee 和 display 间接地访问了基类的私有成员。由于存在同名的 display 成员函数，在派生类中使用基类的公有成员函数 display 时需要显式地指明基类名称，使用形式如下：

```
employee::display();
```

使用基类公有成员函数访问基类私有成员的方法在应用时需要慎重考虑，因为基类的设计者把数据成员声明为私有的，说明设计者想把这些成员对外隐藏起来，不被外界所访问。这种间接访问方式破坏了类设计者的封装意图，可能引入基类私有数据成员被修改的风险。

例 9.6 的继承方式为公有继承，如果改成保护继承和私有继承，程序的运行结果是一样的。

9.1.4 多重继承

前边的继承方式都是单继承，一个派生类只有一个基类。C++支持一个派生类从多个基类继承的方式。

C++多重继承的一般形式为：

```
class 派生类名:[继承方式] 基类名1，[继承方式] 基类名2，…，[继承方式] 基类名n
{
    派生类新增加的成员;
};
```

同单继承一样，继承方式为可选项，包括 public（公有）、private（私有）和 protected（保护）。如果没有显式地指出继承方式，默认继承方式为 private（私有）。

在多重继承中，派生类继承了多个基类的成员，基类中的成员按照继承方式来确定其在派生类中的访问方式。

【例 9.7】 使用多重继承。

假设企业的“技术型员工”兼职做技术的培训工作，现在已有“员工”类和“教师”类，那么可以使用多重继承的方式，让“技术型员工”类同时从“员工”类和“教师”类继承。

```
#include<iostream>
#include<string>
using namespace std;
class employee
{
protected:
   string employeeid;
   string name;
   int age;
public:
void display()
{
```

```
    cout<<"编号:"<< employeeid<<endl;
    cout<<"姓名:"<< name <<endl;
    cout<<"年龄:"<< age <<endl;
}
void setemployee(string inputid,string inputname,int inputage)
{
    employeeid= inputid;
    name= inputname;
    age=inputage;
}
};
class teacher
{
protected:
    string name;
    int age;
    string course;
public:
void display()
{
    cout<<"姓名:"<< name <<endl;
    cout<<"年龄:"<< age <<endl;
    cout<<"所授课程:"<< course<<endl;
}
void setteacher(string inputname,int inputage,string inputcourse)
{
    name= inputname;
    age=inputage;
    course= inputcourse;
}
};
class technican: public employee,public teacher
{
public:
void displaytechnican()
{
    cout<<"编号:"<<employeeid<<endl;
    cout<<"姓名:"<< Employee::name <<endl;
    cout<<"年龄:"<< Employee::age <<endl;
    cout<<"所授课程:"<<course<<endl;
}
void settechnican(string inputid,string inputname,int inputage,string
inputcourse)
{
    employeeid= inputid;
    Employee::name= inputname;
    Employee::age=inputage;
```

```
        course= inputcourse;
    }
    };
    int main()
    {
        technican technican1;
        technican1.settechnican("20100002","李明",26,"C++语言程序设计");
        technican1.displaytechnican();
        return 0;
    }
```

程序运行结果为：

编号：20100002

姓名：李明

年龄：26

所授课程：C++语言程序设计

在例 9.7 中，派生类 Technican 是从 Employee 和 Teacher 两个基类继承而来的，如图9-2 所示，在 Employee 和 Teacher 类中都有 name、age 数据成员，它们都会被派生类继承。当在派生类中访问 name 和 age 成员时，程序会不知道用户到底在访问哪个基类的成员，这就是 **C++多重继承的二义性**。为了消除多重继承的二义性，在派生类中访问两个基类中都有的成员时，需要显式地指明访问的是哪一个基类的成员：

```
cout<<"姓名:"<< employee::name <<endl;
cout<<"年龄:"<< employee::age <<endl;
```

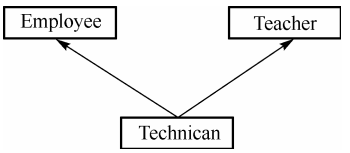


图 9-2　继承关系

9.2　多态与虚函数

9.2.1　多态的概念

多态(polymorphism)是面向对象程序设计的一个重要特征。polymorphism 一词源自希腊文，表示“多种形态”的含义。C++中的多态基于类的继承机制。

下面通过一个例子来阐述多态的特点。假设企业员工类型分为两种，每一种都设计为“员工”类的派生类，设每一个派生类都通过 display 成员函数来输出自己的数据成员，在程序中可以借助 C++的多态机制使用统一的调用方式(使用 display 函数)来分别执行两种输出操作。即实现了使“一个事物”(统一的函数调用方式)呈现“多种形态”(可以执行不同的输出操作)的效果。

【例 9.8】 使用多态。

假设企业员工分为“技术型员工”和“管理型员工”，分别设计“技术型员工”类和“管理型员工”类，它们都继承自员工类，分别改写了基类的 display 函数用做数据成员的输出。

```
#include<iostream>
#include<string>
using namespace std;
class employee
```

```
{
  protected:
    string employeeid;
    string name;
    int age;
  public:
    virtual void display()  //基类中被改写的成员函数声明为虚函数
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
    }
    void setemployee(string inputid,string inputname,int inputage)
    {
        employeeid= inputid;
        name= inputname;
        age=inputage;
    }
};
class technican: public Employee
{
  private:
    string level;           //派生类的数据成员表示岗位级别
  public:
    virtual void display()  //把派生类中与基类的同名函数声明为虚函数
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
        cout<<"岗位级别:"<<level<<endl;
    }
    void settechnican(string inputid,string inputname,int inputage,
    string inputlevel)
    {
        setemployee(inputid,inputname,inputage);
        level=inputlevel;
    }
};
class manager: publicemployee
{
  private:
    string post;                //派生类的数据成员表示职位
  public:
    virtual void display()      //把派生类中与基类的同名函数声明为虚函数
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
```

```
        cout<<"职位:"<<post<<endl;
    }
    void setmanager(string inputid,string inputname,int inputage,string
    inputpost)
    {
        setemployee(inputid,inputname,inputage);
        post=inputpost;
    }
};
int main()
{
    employee *p;
    technican technican1;
    technican1.settechnican("20100001","王军",26,"助理工程师");
    manager manager1;
    manager1.setmanager("20100002","李明",32,"副总经理");
    p=&technican1;
    p->display();
    p=&manager1;
    p->display();
    return 0;
}
```

程序运行结果为：

编号：20100001
姓名：王军
年龄：26
岗位级别：助理工程师
编号：20100002
姓名：李明
年龄：32
职位：副总经理

在例 9.8 中，主函数中首先定义了一个 employee 类的指针，然后分别定义 technican 类的对象 technican1 和 manager 类的对象 manager1，让指针分别指向这两个对象，使用相同的函数调用方式(p->display())来调用它们的成员函数 display，得到了不同的输出结果。这就是C++中的多态。

需要注意在基类 employee 中，display 成员函数声明前增加了 virtual 修饰符，表示这是一个虚函数。虚函数是实现多态的前提条件，如果去掉 virtual 修饰符，重新执行程序，会得到如下结果：

编号：20100001
姓名：王军
年龄：26
编号：20100002
姓名：李明
年龄：32

从结果可以看到，p->display()执行的是基类中的 display 成员函数。当派生类改写了基类中的同名成员函数，且该成员函数不是虚函数时，p 是什么类型的指针，p->display()就会执行该类型中的成员函数 display。此时无论 p 指向哪个派生类对象，所执行的都是基类中的 display 成员函数。

9.2.2　虚函数

从例 9.8 可以看到，虚函数是 C++实现多态的重要条件。当基类中的某个成员函数被声明为虚函数后，派生类中可以改写该函数，实现不同的功能，当使用指向基类的指针指向某一派生类时，通过指针调用该成员函数会执行指针所指向的派生类中的成员函数。

虚函数的使用方法：

(1) 在基类中某个成员函数前加上关键字 virtual，该成员函数被声明为虚函数。

(2) 在派生类中改写该成员函数，改写时使用与基类完全相同的函数声明方式。

(3) 定义一个指向基类的指针，让该指针指向派生类的某一对象。

(4) 通过指针调用该虚函数，所调用的就是指向的派生类中的同名成员函数。

另外，在 C++中只要声明了基类中的虚函数，其派生类中的同名函数会自动成为虚函数。所以在派生类中声明该函数时可以省略 virtual 关键字。

使用虚函数需要注意以下几点：

(1) 虚函数必须是类的非静态成员函数。

(2) 类的构造函数不能定义为虚函数，类的析构函数可以定义为虚函数。

(3) 只需在类中声明虚函数时使用关键字 virtual，在类外进行函数的定义时不需要使用关键字 virtual。

9.2.3　多态的实现机制

在 C++中实现多态需要有以下几个前提条件：

(1) 须存在基类和派生类，即多态依赖类的继承。

(2) 在基类和派生类中具有同名的虚函数。

(3) 对派生类中虚函数的调用必须使用指向基类的指针或引用来进行。

把函数的调用和函数在内存中的地址建立起对应关系的操作称为**关联**(binding)，只有执行关联后，对函数的调用才能找到函数所在内存中的地址，即才能执行该函数中的代码。

关联有两种方式：静态关联和动态关联。在程序的编译期进行的关联操作称为静态关联，在程序运行过程中发生函数调用后才关联到被调用函数内存中的地址的方式称为动态关联。

【例 9.9】 多态与虚函数。

```
#include<iostream>
using namespace std;
class B
{
protected:
    int x;
    int y;
public:
```

```
    virtual void display()
    {
        cout<<"基类 B 的 display 函数被调用! "<<endl;
    }
};
class D1: public B
{
  public:
    virtual void display()
    {
        cout<<"派生类 D1 的 display 函数被调用! "<<endl;
    }
};
class D2: public B
{
  public:
    virtual void display()
    {
        cout<<"派生类 D2 的 display 函数被调用! "<<endl;
    }
};
int main()
{   B b1;
    D1 d1;
    D2 d2;
    cout<<"对象 b1 的大小为:"<<sizeof(b1) <<endl;
    cout<<"对象 d1 的大小为:"<<sizeof(d1)<<endl;
    cout<<"对象 d2 的大小为:"<<sizeof(d1)<<endl;
    B *p;
    p=&d1;
    p->display();
    p=&d2;
    p->display();
    return 0;
}
```

程序运行结果为：

对象 b1 的大小为：12
对象 d1 的大小为：12
对象 d2 的大小为：12
派生类 D1 的 display 函数被调用!
派生类 D2 的 display 函数被调用!

例 9.9 定义了一个基类 B 和两个派生类 D1、D2。B 中有两个整型数据成员。B 中定义了虚函数 display，在派生类 D1 和 D2 中分别进行了改写。通过这个简单的继承结构来分析 C++中是如何实现多态的。

类和对象的大小为其中数据成员大小之和，基类 B 中有 2 个整数成员，D1、D2 继承自 B，

也分别拥有 2 个整数成员。可是从程序运行结果可以看出，三个对象的大小均为 12 字节(32 位整数大小为 4 字节)，多出了 4 字节。

在 C++中，当一个基类中含有虚函数时，编译器在编译时会给每个基类和派生类提供一个虚函数表和一个指针 VPTR，虚函数表中存放着类的虚函数的地址，VPTR 指针位于类中，指向虚函数表中的第一个虚函数。上述三个对象多出来的 4 字节就是 VPTR 指针的大小。图 9-3 表示的是基类对象 b1 的 VPTR 指针和虚函数表。如果基类 B 存在多个虚函数，虚函数表中会存放对应个函数地址，它们的排列顺序与在类中声明的现有次序一致。

主函数中首先声明了一个基类指针变量，然后让该指针指向派生类 D1 的对象 d1，如图 9-4 所示。当执行 p=&d1;语句时，C++编译器进行了隐式类型转换，把 p 由基类指针变量转变成派生类 D1 指针变量；当执行 p->display();语句时，p 根据 d1 的 VPTR 找到对应虚函数表，获取 D1 的 display 函数地址，然后调用执行。

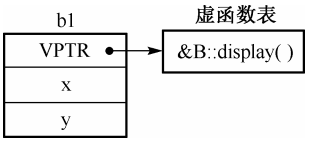


图 9-3　对象 b1 的 VPTR 指针和虚函数表

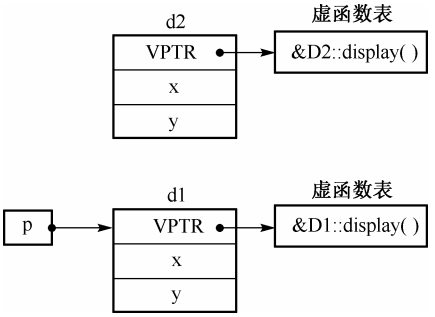


图 9-4　指针 p 指向 d1

同样地，在图9-5中，当执行 p=&d2;语句时，C++编译器通过隐式类型转换，把 p 由派生类 D1 指针变量转变成了派生类 D2 指针变量，然后执行 p->display();语句通过 d2 的 VPTR 找到 D2 的 display 函数并进行调用。

通过以上分析可知，指针 p 是当程序执行到函数调用语句时，通过 VPTR 找到对应的虚函数地址的，所以该函数调用过程是动态关联。C++的多态就是通过虚函数和动态关联机制实现的。

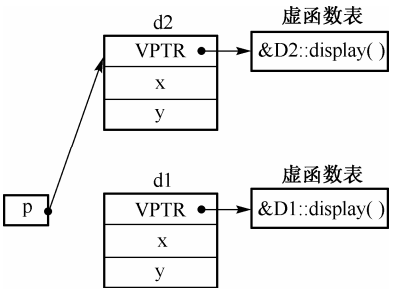


图 9-5　指针 p 指向 d2

9.2.4　纯虚函数与抽象类

在例 9.8 中，基类 Employee 中的 display 虚函数中有数据成员的输出操作，是一个完整的成员函数。可以使用基类 Employee 定义对象，使用 display 成员函数实现对象中数据成员的输出。假设程序中所有的对象都是基于 Employee 的派生类定义的，没有使用基类定义的对象，那么可以把基类的虚函数改成没有任何操作代码的空函数，具体的功能留给派生类改写时根据需要来定义。此时可以把基类中的虚函数声明为**纯虚函数**(pure virtual function)。

纯虚函数指在声明虚函数时被“初始化”为 0 的函数，声明纯虚函数一般形式为：

virtual 函数类型 函数名(参数表列)=0;

当把基类中的虚函数声明为纯虚函数后，就不能使用该基类进行对象定义了。

含有纯虚函数的类称为**抽象类**(abstract class)。抽象类不能建立对象，其作用为给派生类提供统一的函数接口。

【例 9.10】 使用纯虚函数和抽象类。

```
#include<iostream>
#include<string>
using namespace std;
class Employee
{
   protected:
    string employeeid;
    string name;
    int age;
   public:
    virtual void display()=0;
};
class Technican: public Employee
{
   private:
    string level;              //派生类的数据成员
   public:
    virtual void display()     //派生类的成员函数
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
        cout<<"岗位级别:"<<level<<endl;
    }
    void setTechnican(string inputid,string inputname,int inputage,
    string inputlevel)
    {
        employeeid= inputid;
        name= inputname;
        age=inputage;
        level=inputlevel;
    }
};
class Manager: public Employee
{
   private:
    string post;               //派生类的数据成员表示职位
   public:
    virtual void display()     //派生类的成员函数
    {
        cout<<"编号:"<< employeeid<<endl;
        cout<<"姓名:"<< name <<endl;
        cout<<"年龄:"<< age <<endl;
```

```
        cout<<"职位:"<<post<<endl;
    }
    void setManager(string inputid,string inputname,int inputage,string
    inputpost)
    {
        employeeid= inputid;
        name= inputname;
        age=inputage;
        post=inputpost;
    }
};
int main()
{
    Employee *p;
    Technican technican1;
    technican1.setTechnican("20100001","王军",26,"助理工程师");
    Manager manager1;
    manager1.setManager("20100002","李明",32,"副总经理");
    p=&technican1;
    p->display();
    p=&manager1;
    p->display();
    return 0;
}
```

程序运行结果为：

编号：20100001
姓名：王军
年龄：26
岗位级别：助理工程师
编号：20100002
姓名：李明
年龄：32
职位：副总经理

在例 9.10 中，把基类的虚函数重新声明为纯虚函数，此时基类 Employee 成了抽象类，不能定义该抽象类的对象，但可以定义指向抽象数据的指针变量，然后把指针变量指向具体的派生类对象，通过指针调用虚函数来实现多态。

本 章 小 结

继承与多态是面向对象程序设计的重要特征，C++在继承时根据继承方式的不同来确定基类成员在派生类中的访问控制方式。C++同时支持单继承和多继承，C++的多继承容易引起二义性，需要慎重使用。

C++的多态指使用基类指针来引用派生类对象，通过指针调用虚函数的方式来完成对应

派生类对象虚函数的调用。C++实现多态的前提是虚函数的使用。如果不需要使用基类定义对象，可以声明基类中的虚函数为纯虚函数，此时该基类变成抽象类，不能用抽象类定义对象，抽象类用以给继承结构提供一个公有接口。

习 题 9

一、选择题

1. 派生类的对象对它的基类成员中(　　)是可以访问的。

A. 公有继承的公有成员　　B. 公有继承的私有成员

C. 公有继承的保护成员　　D. 私有继承的公有成员

2. 以下有关虚函数的描述，不正确的是(　　)。

A. 基类中的成员函数被声明为虚函数，派生类同名函数自动成为虚函数

B. 虚函数是 C++中实现多态的前提条件

C. 基类和派生类中的虚函数都必须函数声明前使用 virtual 关键字

D. 抽象类中一定有纯虚函数

3. 以下为类 B 和派生类 D 的声明，类 D 中保护的成员(包含数据成员和成员函数)的个数为(　　)。

```
class B
{
   int b1;
public:
   int b2;
   int display() {return b2};
};
class D: protected B
{
  protected:
    int d1;
  public:
    int d2;
    int display() {return d2};
};
```

A. 1　　B. 2　　C. 3　　D. 4

4. 以下程序执行后的输出结果为(　　)。

```
#include <iostream>
using namespace std;
class B
{  public:
   ~B() {cout<<"B";}
};
class D
{  public:
```

```
    ~D(){cout<<"D";}
};
int main()
{
  D d1;
  return 0;
}
```

A. B　　B. D　　C. BD　　D. DB

5. 以下虚函数的定义中不正确的是(　)。

A. `class a{ void virtual f(){};};`

B. `class a{virtual void f(){};};`

C. `class a{ static virtual void f(){};};`

D. `class a{ public: virtual void f(){};};`

二、简答题

1. 简述三种不同继承方式的特点。
2. 简述虚函数的使用原则和方法。

三、分析程序的执行结果

```
#include<iostream >
using namespace std;
class  B
{
int n;
public:
B(){};
B (int a)
    {
        cout << "构造基类" << endl;
        n=a;
        cout << "n="<< n << endl;
    }
~B()  { cout << "析构基类" << endl;  }
};
class D : public B
{
int m;
public:
D(int a, int b): B(a)
    {
        cout << "构造派生类" << endl;
        m=b;
        cout << "m=" << m << endl;
}
    ~D(){ cout << "析构派生类" << endl;  }
};
```

```
    int main ()
{
    D d(1,2);
    return 0;
}
```

四、程序设计题

建立一个基类圆(circle)，具有数据成员“半径”(radius)和成员函数“面积”(area())。然后使用基类圆创建派生类球(ball)，该派生类具有数据成员“半径”(radius)和成员函数“体积”(volume())，建立这两个类调用其成员函数。要求：

(1) 使用类的构造函数实现对象的初始化。

(2) 使用公有函数实现对象的初始化。

第 10 章　C++流与文件操作

10.1　C++流的概念

C++语言中“流”是指数据的流动。“流”既可以表示数据从内存传送到某个载体或设备中，即输出流；也可以表示数据从某个载体或设备传送到内存缓冲区变量中，即输入流。数据的输入和输出(简称为 I/O)包括以下两方面的内容：

(1) 标准的输入输出(简称标准 I/O)，即从键盘输入数据，从屏幕输出数据。

(2) 文件的输入输出(简称文件 I/O)，即从存储介质上的文件输入数据，然后将结果输出到外存储介质。

10.2　输入/输出标准流类

C++系统在内存中为每个数据流开辟了一个缓冲区用于存放流中的数据。如编程中经常用 cin 和 cout 语句进行输入和输出，用“>>”运算符将数据从键盘缓冲区输入到内存中的变量；用“<<”运算符将缓冲区中的全部数据传送到显示器显示。

VC++ 2005 系统提供了功能强大的 I/O 流类库，可使用不同的类去实现各种功能。

10.2.1　C++中的I/O流库

1．I/O库中常见的流类有：

(1) ios：为根基类，它直接派生 4 个类：输入流类 istream、输出流类 ostream、文件流基类 fstreambase 和字符串流基类 strstreambase。

(2) istream：通用输入流类，支持输入操作。同时继承了输入流类和文件流基类。

(3) ostream：通用输出流类，同时继承了输出流类和文件流基类。

(4) iostream：通用输入输出流类，由类 istream 和类 ostream 派生，支持输入输出操作。

(5) ifstream：输入文件流类，由类 istream 派生，支持输入文件操作。

(6) ofstream：输出文件流类，由类 ostream 派生，支持输出文件操作。

(7) fstream：输入输出文件流类，由类 iostream 派生，支持输入输出文件操作。

I/O 库常用的流类继承关系如图10-1所示。

10.2.2　标准输入/输出流对象

1．输入流对象

C++语言有 3 种输入流对象的操作方式。

(1) cin 是类 istream 的对象，用于从标准输入设备获取数据，使用运算符“>>”将输入的数据传送给程序中的变量。“cin>>”除可以输入数据外，也可以输入字符；

(2) 用 get()函数输入单个字符，一般的使用方式为：

输入流对象.get();

该函数返回输入的字符，若遇到输入流中的结束符，则函数返回文件结束标志 EOF(End of File)。

(3) 用 getline()函数读入字符串，一般使用格式为：

输入流对象.getline(字符指针，字符个数 n)

如果输入的字符串中字符个数小于 n，则字符指针指向字符串实际输入的字符，如果输入的字符串个数大于等于 n，则指针指向存储位置为 n–1 的字符。

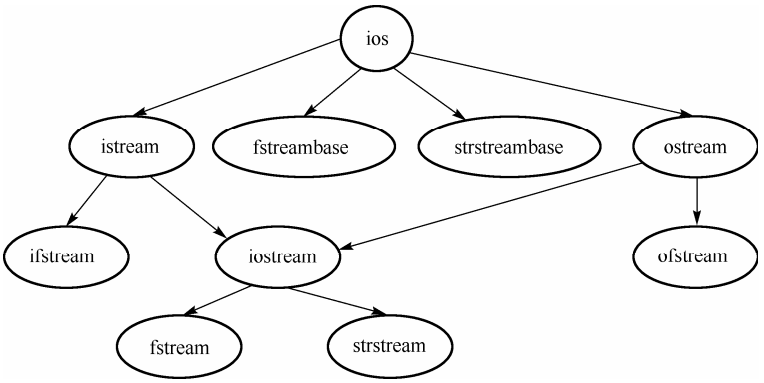


图 10-1 I/O 库中类的继承关系图

【例 10.1】 输入流对象操作符应用举例。

```
#include<iostream>
using namespace std;
void main()
{
    int x,y;
    char ch1;
    cout<<"输入整数给 x,y；输入一个字符给 ch1"<<endl;
    cin>>x>>y;
    ch1=cin.get();
    cout<<"x,y 的值为:"<<x<<"  "<<y<<endl;
    cout<<"输入的字符为:";
    cout<<ch1;
    cout<<endl;
}
```

运行程序后屏幕显示：

输入整数给 x, y；输入一个字符给 ch1

从键盘输入 3 5 h 回车

输出结果为：

x,y 的值为：3 5

输入的字符为：h

【例 10.2】 用 getline()输入字符串应用举例。

```
#include<iostream>
using namespace std;
void main()
{    char ch1[10];
     cout<<"从键盘输入一个少于 10 个字符的字符串:";
     cin.getline(ch1,10);
     cout<<ch1<<endl;
}
```

程序运行后屏幕显示：

从键盘输入一个少于 10 个字符的字符串:

从键盘输入如一行字符：I Love China!

运行结果为：I Love Ch

注意：当连续输入多个数据时，各数据之间应使用空格隔开，输入的数据类型应与定义的变量的类型一致，使用 get()提取符号时，不能使用空格，否则输入的空格也被当做输入字符对待。

2．输出流对象

在 C++语言中提供了 2 种输出流对象的操作方式：cout>>和 put()操作方式。

(1) cout 是输出流类 ostream 的对象，输出结果流向标准的输出设备显示器。其应用方式在前面的章节中都使用过，在此主要介绍输出流格式控制。

在前面章节中大多数的题例没有指定输出格式，由系统根据数据类型选择默认的格式，但用户有时希望数据能按照指定的格式输出，如保留小数 2 位、左对齐或右对齐等。

输出格式状态是在类 ios 中定义的枚举值，用于指定输出数据的格式。所以在引用时应加上类名 ios 和作用域运算符“::”，常用的输出格式状态见表 10-1。

表 10-1　常用输出格式状态

输出格式状态	功　能	输出格式状态	功　能
ios::left	按左对齐输出	ios::scientific	以科学记数法格式输出
ios::right	按右对齐输出	ios::fixed	以定点格式输出
ios::skipws	跳过从当前位置开始的空白符	ios::showbase	在数据输出前面加“基指示符”,如 ox 表示十六进制
ios::dec	按十进制数输出	ios::showpoint	强制输出的浮点数中带有小数点和无效数字 0
ios::oct	按八进制数输出	ios::pos	使输出的数值前带有正号“+”
ios::hex	按十六进制数输出	ios::uppercase	使输出的浮点数中使用的字母为大写

【例 10.3】 输出格式控制符应用举例。

```
#include<iostream>
using namespace std;
void main()
{
double a=13.5,b=3.14259,c=-123.657;
cout<<a<<"  "<<b<<"  "<<c<<endl;
cout.setf(ios::left );          //设置按左对齐方式输出
```

```
    cout.fill('*');                    //设置填充字符为*
    cout<<"a 的值为:";
    cout.width(10);                    //设置输出下一个数据值的域宽为 10
    cout<<a<<endl;
    cout.unsetf(ios::left);            //终止左对齐方式
    cout.setf(ios::right);             //设置按右对齐方式输出
    cout<<"b 的值为:";
    cout.width(10);                    //设置输出下一个数据值的域宽为 10
    cout<<b<<endl;
    cout.setf(ios::showpoint);         //强制显示小数点
    cout<<a<<"  "<<b<<"  "<<c<<endl;
    cout.unsetf(ios::showpoint);       //恢复默认格式输出
    cout<<a<<"  "<<b<<"  "<<c<<endl;
    }
```

此程序的运行结果为：

```
3.14259  –123.657
a 的值为：13.5******
b 的值为：***3.14259
13.5000   3.14259   –123.657
13.5   3.14259   –123.657
```

输出流控制符是在头文件 iomanip 中定义的，因此要使用下面的输出流控制符，应在程序中加上头文件#include<iomanip>。常用输出流控制符见表 10-2。

表 10-2 常用输出流控制符

控制标示符	功能说明
.dec	转换为按十进制输出的整数
.oct	转换为按八进制输出的整数
.hex	转换为按十六进制输出的整数
.ws	从输入流中一次性读取所有连续的空白符
.endl	输出换行符'\0'并刷新流
.ends	输出一个空字符'\0'
.flush	刷新一个输出流
.setw(int n)	设置下一个数据的输出域宽度为 n
.setiosflags(long f)	设置 f 对应的格式化标志，功能与 setf(long f)相同

【例 10.4】 输出流控制举例。

```
    #include<iostream>
    #include<iomanip>
    using namespace std;
    void main()
    {
    int a=10,b=20,c=2000;
    cout<<a<<"  "<<b<<"  "<<c<<endl;
    cout<<oct<<a<<"  "<<b<<"  "<<c<<endl;   //按八进制输出
    cout<<hex<<a<<"  "<<b<<"  "<<c<<endl;   //按十六进制输出
```

```
cout<<setw(10)<<a<<setw(10)<<b<<setw(10)<<c<<endl;
cout<<setiosflags(10);                    //设置当前进位计数制的标志
cout<<setw(10)<<a<<setw(10)<<b<<setw(10)<<c<<endl;
cout<<dec<<a<<"  "<<b<<"  "<<c<<endl;    //按十进制输出
}
```

程序的运行结果为：

```
10   20   2000
10   24   3720
a   14   7d0
          a          14          7d0
        0xa        0x14        0x7d0
10   20   2000
```

另外，ios 类提供公有的成员函数对流的状态进行检测和输入输出格式设置等，表 10-3 给出了每个成员函数的格式和功能说明。

表 10-3　常用 ios 类中的成员函数表

成员函数名及格式	功 能 说 明
int bad();	操作出错时返回非 0 值
int eof()	读取到文件最后结束符时返回非 0 值
int fail();	操作失败时返回非 0 值
void clear();	清除 bad、eof、fail 对应的标志状态，恢复为 0 值
char fill();	返回当前使用的填充字符
char fill(char c);	重新设置流中用于输出数据的填充字符为字母 c
long flags(long f);	重新设置格式状态字为 f 的值
int good()	操作正常时返回非 0 值，否则返回 0
int precision(n)	省略 n 时为返回浮点数输出精度，即有效数字的位数
int restate()	good()的反函数
int width(n)	省略 n 时，返回当前的输出域宽度，n 为设置的输出域宽度

（2）用输出流类成员函数 put()输出字符

C++中如果想输出单个字符，除可以使用“cout<<”语句外，还可以使用输出流函数 put()，其一般格式为：

```
输出流对象.put(ch);
```

其中 ch 为要输出的字符。

【例 10.5】 用输出流成员函数 put()输出单个字符。

```
#include<iostream>
using namespace std;
void main()
{ char str[]="I Love China!";
for(int i=0;i<strlen(str);i++)
cout.put(str[i]);
cout<<endl;
}
```

程序运行结果为：

I Love China!

10.3 文件操作

文件是一组相关数据的有序集合。文件被保存到磁性介质上，按一定的组织形式存储。每个文件都对应一个文件名。文件名由文件主名和扩展名组成，中间用圆点分开。

文件主名是一个有效的 C++标识符，扩展名一般由 1～3 个字符组成，利用扩展名可以区分文件的类型。如在 C++系统中，用.cpp 表示程序文件，用.obj 表示目标文件，用.exe 表示可执行文件。对于用户建立的用于保存数据的文件，通常扩展名用.dat 表示，由字符构成的文本文件则用.txt 作为扩展名等。

从用户角度看，文件可分为普通文件和设备文件两种。

普通文件是指存储在磁盘或外部存储介质上的数据文件，可以是源文件、目标文件、可执行文件等。设备文件是指与主机相连的各种外部设备，如显示器、打印机、键盘等。在操作系统中，把外部设备也当做文件来进行管理。通常把显示器作为标准的输出设备，把键盘作为标准的输入设备，磁盘既可以作为输入设备，也可以作为输出设备。

从文件编码方式的角度看，可将文件分为两种类型，一种为字符格式文件，又称 ASCII 码文件或文本文件；另一种为内部格式文件，又称二进制文件或字节文件，二进制文件可更好地节省存储空间和转换时间。

C++中的文件流实际上就是以外存文件为输入输出对象的数据流。输入文件流是指从外存文件流向内存的过程，输出文件流是指从内存流向外存的过程。

C++对文件流分为 3 类：输入流、输出流及输入/输出流。其中:

(1) ifstream 流类，是从 istream 类派生的，用于外存文件的输入操作；

(2) ofstream 流类，是从 ostream 类派生的，用于外存文件的输出操作；

(3) fstream 流类，是从 iostream 类派生的，用于外存文件的输入和输出操作。

要对外部文件进行操作，应先定义一个文件流类的对象，然后通过文件流对象操作数据。

定义流文件对象的一般格式为：

```
fstream  流对象名;
```

10.3.1 文件的打开与关闭

1. 打开文件

要对外存文件进行操作，首先要在开始包含#include<fstream>命令，因为该文件包含了支持文件读写的 IO 流类的定义。

文件在进行读写操作前，应先打开，其目的是为文件流对象和特定的外存文件建立关联，并指定文件的操作方式。

文件打开方式的一般格式为：

```
文件流对象名.open("文件名", 文件打开模式)
```

其中：文件名可包含路径说明；文件打开模式，可为表 10-4 中的文件操作方式之一。

在默认情况下，以文本方式打开文件，若需要以二进制方式打开文件，则需要将打开方式设置为“binary”。

文件的操作方式用于说明是输入操作还是输出操作，是对 ASCII 码操作还是对二进制文件操作等。C++中提供的打开文件方式如表 10-4 所示。

表 10-4　文件的打开模式(mode)

文件操作方式	功能
ios::in	打开文件进行读操作，如果文件不存在则出错
ios::out	打开文件进行写操作，如果文件不存在，则建立一个文件，否则将清空文件，该方式为默认方式
ios::ate	打开文件后，指针定位到文件尾部
ios::app	以追加方式打开文件，所有追加内容都在文件尾部进行
ios::trunc	如果文件已存在则清空原文件，否则创建新文件
ios::binary	打开二进制文件(非文本文件)

每个被打开的文件，其内部都有一个文件指针，指向当前要进行读写操作的位置。每读出一个字符，指针自动后移一字节。当指针指向文件尾时，将遇到文件结束符 EOF，此时用 eof()函数检测，得到非 0 的一个数值，表示文件结束了。

2．关闭文件

当对一个文件的读写操作完成后，为了保证数据安全，切断文件与流的联系，应及时关闭文件。关闭文件的一般格式为：

```
流对象名.close( )
```

注意：如果未指明以二进制方式打开文件，则默认是以文本方式打开文件。对于 ifstream 流，mode 参数的默认值为 ios::in；对于 ofstream 流，mode 的默认值为 ios::out。

10.3.2　文本文件的读写操作

文本文件的读写操作分为顺序读写和随机读写两种。

1．顺序读写文件

顺序读写是指从文件头一直读或写到文件尾，通常采用 get()、getline()、put()、read()或 write()等函数来完成对文件的读写操作。

【例 10.6】 用文件流对象，将整型数组 a[10]中的数据写入到指定的文本文件中，然后再将这个文件中的数据读入内存，并显示在屏幕上。

```
#include<iostream>
#include<fstream>                  //文件流操作的预编译处理命令
using namespace std;
void main()
{ int a[10]={3,5,6,2,13,45,67,23,44,55},x;
fstream f;                         //定义流对象名
f.open("myfile.txt",ios::out);  //以输出方式打开文件，即建立一个文本文件
if(f.fail())
```

```
{ cout<<"打开文件失败"<<endl;exit(1);}    //打开文件失败退出程序
for(int k=0;k<10;k++)
  f<<a[k]<<" ";                  //将数据输出到当前路径下的 myfile 文本文件中
f.close();                       //关闭文件
f.open("myfile.txt",ios::in);    //以输入方式打开文件
if(f.fail())
{ cout<<"打开文件失败"<<endl;exit(2);}    //打开文件失败退出程序
while(!f.eof())                          //从文件中输入数据到 x
{f>>x;
cout<<x<<"  ";
}
cout<<endl;
f.close();
}
```

程序运行结果为:

3, 5, 6, 2, 13, 45, 67, 23, 44, 55

程序中 f<<a[k]<<" ";语句用于在各个数据后加一个空格，若没有空格，则文本文件中所有数据之间无分隔符。

【例 10.7】从键盘上输入若干行文本字符到 dat 文件中,直到按下 Ctrl+Z 组合键为止(此组合键代表文件结束符 EOF)。

```
#include<iostream>
#include<fstream>
using namespace std;
void main(void)
{ char ch;
ofstream f2( "wr2.txt");
//定义 f2 为流对象名，并在当前目录下建立(默认打开)一个 wr2 的文本文件
if(!f2){cerr<<"file of wr2.dat not open!"<<endl;
exit(1);}
ch=cin.get();           //从 cin 字符中提取一个字符到 ch 中
while(ch!=EOF){
    f2.put(ch);         //把 ch 字符写入到 f2 流中
    ch=cin.get();}
f2.close();
}
```

此程序运行后，将在当前目录下建立一个 wr2.txt 的文本文件。

【例 10.8】 用 get()函数，将例10.7 中 wr2.dat 文件的字符串读入内存，并依次显示到屏幕上，统计出字符串的个数。

```
#include<iostream>
#include<fstream>
using namespace std;
void main(void)
{ ifstream f2("wr2.txt",ios::in);      //以默认方式 open 一个文本文件
if(!f2){cout<<"文件打不开"<<endl;exit(1);}
```

```
    char ch;
    int i=0;
    while(f2.get(ch))
    {cout<<ch;
    if(ch=='\n')i++;
    }
    cout<<endl<<"字符串的个数是:"<<i<<endl;
    f2.close();
    }
```

程序的运行结果是将 wr2.dat 中的字符串读入 ch 变量中，并按行输出，最后输出字符串的个数。

【例 10.9】 用 getline()函数，从文本文件中每次读入一行字符，并依次显示到屏幕上。

```
    #include<iostream>
    #include<fstream>
    using namespace std;
    void main(void)
    { ifstream f2;      //定义文件流对象 f2
    char s[200],fname[20];
    cout<<"请输入文本文件名";
    cin>>fname;
    f2.open(fname);     //打开指定的文件
    if(f2.fail())
    {cout<<"打开文件失败"<<endl;
    exit(1);
    }
    f2.getline(s,200);  //从文件中读入一行字符
    while(!f2.eof())
    {cout<<s<<endl;
    f2.getline(s,200);
    }
    f2.close();
    }
```

该程序运行后在屏幕上显示：

“请输入文本文件名”

从键盘输入：wr2.txt 回车。

将按行显示 wr2.txt 中的所有字符串内容。

10.3.3　二进制文件的读写操作

二进制文件不同于文本文件，它可用于任何类型的文件(包括文本文件)。存储到二进制文件中的字符不做任何转换就可保存到外存文件中。

一般地，对二进制文件的读写可采用两种方法：一种是使用 get()和 put()；另一种是使用 read()和 write()。

1．用read()和write()读写二进制文件

对二进制文件的读写主要用文件流类成员函数 read()和 write()来实现，这两个成员函数的一般格式为：

```
文件流对象.read(字符指针 buffer , 长度 len);
文件流对象.write(字符指针 buffer , 长度 len);
```

其中字符指针 buffer 用于指向内存中的一块存储区域，长度 len 指读写数据的字节数。

【例10.10】 将例10.6 整型数组a[10]中的数据写入到二件制文件中，然后再将这个文件中的数据读入内存，并显示在屏幕上。

```
#include<iostream>
#include<fstream>                                  //文件流操作的预编译处理命令
using namespace std;
void main()
{ int a[10]={3,5,6,2,13,45,67,23,44,55},x;
fstream f;                                         //定义流对象名
f.open("myfile.dat",ios::out|ios::binary); //以输出方式打开一个二进制文件
if(f.fail())
{ cout<<"打开文件失败"<<endl;exit(1);}             //打开文件失败退出程序
for(int k=0;k<10;k++)
  f.write((char*)&a[k],sizeof(int)); //将数据输出到当前路径下的 myfile 文件中
f.close();                                 //关闭文件
f.open("myfile.dat",ios::in|ios::binary);   //以输入方式打开文件
if(f.fail())
{ cout<<"打开文件失败"<<endl;exit(2);}             //打开文件失败退出程序
f.read((char*)&x,sizeof(int));                     //从文件中输入数据到 x
while(!f.eof())
{
cout<<x<<"  ";
f.read((char*)&x,sizeof(int));
}
cout<<endl;
f.close();
}
```

程序运行后，将在屏幕上显示：3, 5, 6, 2, 13, 45, 67, 23, 44, 55

本例中，要写入的数据是整型，而 read()和 write()函数中的第一个参数要求为字符指针，因此使用了(char*)&a[k]和(char*)&x 进行强制类型转换。

在程序中：

```
for(int k=0;k<10;k++)
  f.write((char*)&a[k],sizeof(int));
```

这两条语句的作用是将数据一个一个地写入到文件中，如果将这两条语句合并为如下的一条语句：

```
f.write((char*)&a[0],sizeof(a));
```

则可以将 a 数组中的数据一次性写入到文件中，效率更高。

2．用put()和get()读写二进制文件

【例10.11】 将一字符串写入到二进制文件 mybinary.dat 中，然后再依次读取二进制文件中的字符，并显示在屏幕上，最后输出数字字符的个数。

```
#include<iostream>
#include<fstream>           //文件流操作的预编译处理命令
using namespace std;
void main()
{ char ch;
int n=0;
fstream f1;
f1.open("mybinary.dat",ios::out|ios::binary);
if(!f1)
{   cout<<"mybinary.dat can't opan"<<endl;
    exit(1);
}
ch=cin.get();               //从 cin 流中提取一个字符给 ch
while(ch!=EOF)              //输入字符串，按 Ctrl+Z 结束
{   f1.put(ch); ch=cin.get();}
f1.close();
f1.open("mybinary.dat",ios::in|ios::binary);
if(!f1)
{   cout<<"mybinary.dat can't opan"<<endl;   exit(1);}
while(!f1.eof())            //读入字符串，并输出在屏幕上
{f1.get(ch);
cout<<ch;
if(ch>='0'&& ch<='9')n++;   //统计数字字符的个数
}
cout<<"数字字符的个数是:"<<n<<endl;
f1.close();
}
```

运行该程序后在屏幕上输入：

abcdef 1234 gh，回车

按 Ctrl+Z 结束字符串的输入。则屏幕上显示：

abcdef 1234 gh

数字字符的个数是：4

3．用与文件指针有关的流成员函数实现对二进制文件的访问

在 C++中，二进制文件的读写操作允许用指针进行控制，目的是将指针移动到指定的位置后，再对文件进行读写操作。文件流提供了如表10-5所示的常用文件指针成员函数。

表 10-5　文件流与文件指针有关的成员函数

文件操作方式	功能
seekg(位置)	将输入位置指针移动到指定位置
seekg(位移量,参考位置)	以参照位置为基础移动指定的位移量
seekp(位置)	将输出位置指针移动到指定位置
seekp(位移量,参考位置)	以参照位置为基础移动指定的位移量
tellg()	返回输入文件指针的当前位置
tellp()	返回输出文件指针的当前位置

表10-5中函数名的最后一个字符分别为 p 和 g，分别代表 put 和 get。参数中的位置和位移量均为长整型，以字节为单位。“参照位置”可以是:

```
ios::beg    //表示文件头，为默认值
ios::cur    //当前位置
ios::end    //文件尾
```

例如:

```
f1.seek(100);                    //表示将文件指针移动到第 100 字节位置
f1.seek(50,ios::cur)             //表示将文件指针从当前位置前移 50 字节
f1.seek(-50,ios::cur)            //表示将文件指针从当前位置后移 50 字节
```

【例 10.12】 编一程序，将随机函数产生的 10 个整数值排序后存储到磁盘文件中，然后再从文件中读入排序后的数据，并在屏幕上显示出来。

```
#include<iostream>
#include <ctime>                 //调用系统时间
#include<cstdlib>                //调用随机数的函数
#include<fstream>
using namespace std;
void main()
{ int a[10];
int i,j,k,x;
srand((unsigned)time(0));        //使每次产生的随机数不同
  for(i=0;i<10;i++)
      a[i]=rand()%100;           //保证产生的随机数在以内
  for(i=0;i<9;i++)               //排序
      for(j=i+1;j<10;j++)
          if(a[i]<a[j]){k=a[i];a[i]=a[j];a[j]=k;}
  ofstream f1("internum.dat",ios::out|ios::binary); //建立二进制文件
  if(!f1){cout<<"文件打不开! "<<endl;exit(1);}
  for(int i=0;i<10;i++)
  f1.write((char*)&a[i],sizeof(int));        //将 a 数组中的数据写入文件
  f1.close();                                //关闭文件
  ifstream f2("internum.dat",ios::in|ios::binary);  //打开文件
  if(!f2){cout<<"文件打不开! "<<endl;exit(1);}
  for(i=0;i<10;i++)                          //依此读出数据
  { f2.read((char*)&x,sizeof(int));
  cout<<x<<" ";                              //在屏幕上显示数据
  }
cout<<endl;
f2.close();
}
```

程序运行后，屏幕上将显示排序后的 10 个数据。

10.4 应用举例

【例 10.13】 学生成绩综合管理。

分析：学生成绩综合管理应包括如下的内容。

(1) 建立数据档案。从键盘输入若干条学生记录到指定的文件中，例如：

g:\student\student\xscjgl.dat

(2) 向文件尾追加一条记录；从文件中查找给定姓名的记录，并返回查找信息；读入文件中的多条记录，并输出到屏幕上。

学生记录一般应包括学生姓名 name 和学生成绩 grade 两个字段。

(1) 建立数据档案(假设有 5 条记录)

```
#include<iostream>
#include<fstream>     //文件流操作的预编译处理命令
#include<string>
using namespace std;
struct stu
{  char name[10];
   int graed;
};
void main()
{  char *p="g:\\student\\student\\xscjgl.dat";
   fstream fout(p,ios::out|ios::trunc|ios::binary);
   if(!fout){cout<<"文件没打开,退出\n";exit(1);}
   stu x;
   cout<<"请输入学生记录，按 Ctrl+Z 结束输入:"<<endl;
   while(cin>>x.name)
   {cin>>x.graed;
   fout.write((char*)&x,sizeof(x));
   }
   fout.close();
   cout<<"输入过程结束! "<<endl;
}
```

程序运行后，显示：

请输入学生记录，按 Ctrl+Z 结束输入：

然后从键盘输入如下内容：

张静 90

李莉新 78

章胜利 85

齐鑫 93

赵明明 87

^z

输入过程结束!

(2) 向文件尾追加一条记录；并从文件中查找给定姓名的记录，并返回查找信息，最后读入文件中的多条记录，并输出到屏幕上。

```
#include<iostream>
#include<string>
#include<fstream>
using namespace std;
```

```
struct stu{
    char name[10];
    int grade;
};
void append(fstream& fio,int &n,const stu& rec)
{  fio.seekp(0,ios::end);
   fio.write((char*)&rec,sizeof(rec));
   n++;
}
bool find(fstream&fio,int n,stu& rec)
{  fio.seekg(0);
   stu x;
   for(int i=0;i<n;i++)
   {fio.read((char*)&x,sizeof(x));
   if(strcmp(x.name,rec.name)==0)
   {cout<<"找到姓名为"<<x.name<<"的记录，成绩为:"<<x.grade<<endl;
   rec=x;
   return true;
   }
   }
   cout<<"没有找到姓名为"<<rec.name<<"的记录"<<endl;
   return false;
}
bool update(fstream& fio,int n,const stu&rec)
{  fio.seekg(0);
   stu x;
   for(int i=0;i<n;i++){
   fio.read((char*)&x,sizeof(x));
   if(strcmp(x.name,rec.name)==0){
        fio.seekg(-sizeof(x),ios::cur);
        fio.write((char*)&rec,sizeof(x));
        cout<<rec.name<<"的记录被修改! "<<endl;
        return true;
        }
   }
   cout<<"没有找到姓名为"<<rec.name<<"的记录! "<<endl;
   return false;
}
void print(fstream&fio,int n)
{  fio.seekg(0);
   stu x;
   for(int i=0;i<n;i++){
   fio.read((char*)&x,sizeof(x));
   cout<<x.name<<" "<<x.grade<<endl;
   }
}
void main(void)
{  char *p="g:\\student\\student\\xscjgl.dat";
   fstream f1(p,ios::in|ios::out|ios::binary);
```

```
        if(!f1){cout<<"没有打开文件\n";exit(1);}
        stu x;
        int k;
        f1.seekg(0,ios::end);
        int n=f1.tellg()/sizeof(x);
        while(1){
        cout<<"学生成绩综合管理系统"<<endl<<endl;
        cout<<"1.追加一条记录"<<endl;
        cout<<"2.按学生姓名查找"<<endl;
        cout<<"3.按姓名修改学生记录"<<endl;
        cout<<"4.输出文件中的所有学生记录"<<endl;
        cout<<"5.结束程序的运行"<<endl;
        cout<<"请输入您的选择:"<<endl<<endl;
        cin>>k;
        switch(k){
    case 1:
        cout<<"输入要追加的学生的记录:";
        cin>>x.name>>x.grade;
        append(f1,n,x);
        break;
    case 2:
        cout<<"输入要查找的学生的记录:";
        cin>>x.name;
        find(f1,n,x);
        break;
    case 3:
        cout<<"输入要修改的学生的记录:";
        cin>>x.name>>x.grade>>x.grade;
        update(f1,n,x);
        break;
    case 4:
        cout<<"学生档案中的全部记录如下:"<<endl;
        print(f1,n);
        break;
    case 5:
        cout<<"程序运行结束，谢谢使用！欢迎再来！！"<<endl;
        return;
        }}
    f1.close();
    }
```

程序运行后，将在屏幕上显示 1～5 项功能，供用户选择。如选择 1，则追加记录，如输入“程莉敏 97”然后再输入 4，输出文件中的所有学生记录，则屏幕上显示：

张静 90
李莉新 78
章胜利 85
齐鑫 93
赵明明 87
程莉敏 97

本 章 小 结

1. C++流包括标准的 I/O 流、文件流和字符串流三种。标准 I/O 流用于对常用设备的输入和输出，即键盘和显示器，文件 I/O 流用于对磁盘上的文件进行输入和输出，字符串流用于对内存中的字符和字符数组以文件的形式进行输入和输出的操作。

2. 对文件流来说，无论是输入还是输出，都可以利用相同的成员函数和格式标示符进行处理。C++中可以建立两种类型的数据文件，即字节文件和二进制文件。对数据文件的访问方式包括：输入输出方式和追加方式。

3. 要使用一个数据文件，第一步是在文件开始写一个预处理命令#include<fstream>；第二步是定义一个数据流对象变量，如 fstream f1；第三步是利用 open 命令打开(或建立)一个数据文件；第四步是利用 read()、write()、get()、put()、getline()命令对文件进行读写操作；第五步是关闭文件，如 f1.close()。

习 题 10

一、选择题

1. 进行文件操作时，需要包含一个(　　)文件。
   A. iostream　　B. String　　C. fstream　　D. cstdlib
2. 使用函数 setw()对数据进行格式控制输出时，应该包含(　　)文件。
   A. iostream　　B. string　　C. fstream　　D. iomanip
3. 格式化输入输出的控制符中，下面(　　)是设置域宽的。
   A. ws　　B. setw()　　C. setfill　　D. oct
4. 利用 ifstream 流类定义一个流对象打开一个文件时，文件的隐含打开方式是(　　)。
   A. ios::binary　　B. ios::out　　C. ios::trunk　　D. ios::in

二、判断程序的运行结果或指出程序功能

1.

```
#include<iostream>
#include<strstream>
using namespace std;
void main()
{  char a[]="35 12 67 47 -10 -1";
   istrstream str(a);
   int n;
   str>>n;
   while(n!=-1)
   {cout<<dec<<n<<" "<<oct<<n<<" "<<hex<<n<<endl;
   str>>n;
   }
}
```

2.

```
#include<iostream>
using namespace std;
void main( )
{ cout.fill('*');
  cout.width(10);
  cout<<123.56<<endl;
  cout.width(6);
  cout<<123.56<<endl;
  cout.width(2);
  cout<<123.56<<endl;
}
```

3.

```
#include<iostream>
#include<fstream>
using namespace std;
void fun1(char* fname)
{  ofstream fout(fname);
   char a[20];
   cout<<"输入一个字符串少于个字符"<<endl;
   while(1)
   {   cin>>a;
   if(strcmp(a,"end")==0)break;
   fout<<a<<endl;
   }
   fout.close();
}
void main()
{  char *p="g:\\student\\abc.dat";
   fun1(p);
}
```

4.

```
#include<iostream>
#include<fstream>
using namespace std;
void main()
{  ofstream ostrm;
   ostrm.open("f1.txt");
   ostrm<<120<<endl;
   ostrm<<310.85<<endl;
   ostrm.close();
   ifstream istrm("f1.txt");
   int n;
   double d;
   istrm>>n>>d;
   cout<<n<<','<<d<<endl;
   istrm.close();
}
```

5.

```
#include<iostream.h>
#include <fstream.h>
#include<stdlib.h>
void main()
{
   fstream infile;
   infile.open("f2.dat",ios::in);
   if(!infile)
   {
   cout<<"f2.dat cannt open.\n";
   abort();
   }
   char s[80];
   while (!infile.eof())
   {
   infile.getline(s,sizeof(s));
   cout<<s<<endl;
   }
   inflie.close();
}
```

6.

```
include<iostream>
#include<fstream>
#include<cstdlib>
using namespace std;
struct person
{  char name[20];
   double height;
   unsigned short age;
};
person people[40]={ "Wang",1.65,25,"Zhang",1.74,24,"Li",1.89,21,"Hang",
1.70,22};
void main()
{  fstream infile,outfile;
   outfile.open("f5.dat",ios::out|ios::binary);
   if(!outfile)
   {  cout<<"f5.dat cannt open.\n"; abort(); }
   for(int i=0;i<4;i++)
   outfile.write((char *)&people[i],sizeof(people[i]));
   outfile.close();
   infile.open("f5.dat",ios::in|ios::binary);
   if(!infile)
   {  cout<<"f5.dat cannt open.\n"; abort(); }
   for(int i=0;i<4;i++)
   {  infile.read((char *)&people[i],sizeof(people[i]));
   cout<<people[i].name<<" "<<people[i].height<<" "<<people[i].age<<endl;
   }
   infile.close();
}
```

第 11 章　VC++ 2005 应用程序开发实例

20 世纪 90 年代中期以后，随着 Windows 操作系统在全球的盛行，GUI(图形用户界面)应用程序设计也在全球领域内风靡起来。计算机多媒体技术、图形图像技术、计算机通信与网络技术的发展，使应用程序设计也需要有强大的可视化设计工具来支持。Microsoft 公司推出的支持可视化编程 VC++ 2005 的集成环境就是这其中的典型代表，在进入新世纪之后推出的 VC++ 2005\VC++ 2008 更是兼容了以前 Visual C++ 6.0 的集成化开发环境。

MFC(Microsoft Foundation Classes)，是微软公司提供的类库(classlibraries)，以C++类的形式封装了 Windows 的 API，并且包含一个应用程序框架，以减少应用程序开发人员的工作量。

本章的内容主要针对 VC++ 2005 中 MFC 编程方法进行讨论。首先介绍如何用 MFC AppWizard(应用程序向导)来生成并建立应用程序的基本框架，然后讨论应用程序基本框架的构成，以及基于 MFC 类库的应用程序的执行，最后介绍了两个编程实例，希望可以帮助大家更快地学习并掌握 VC++ 2005 的编程方法。

11.1　MFC应用程序

11.1.1　创建应用程序

VC++ 2005 提供的应用程序向导能自动生成应用程序的标准框架，该框架定义了程序的基本结构，大大减轻了编程的工作量，起到事半功倍的效果。

按照之前讲述的方法，启动 Visual Studio 2005 C++，选择菜单“文件”→“新建”→“项目”。弹出界面如图11-1所示。

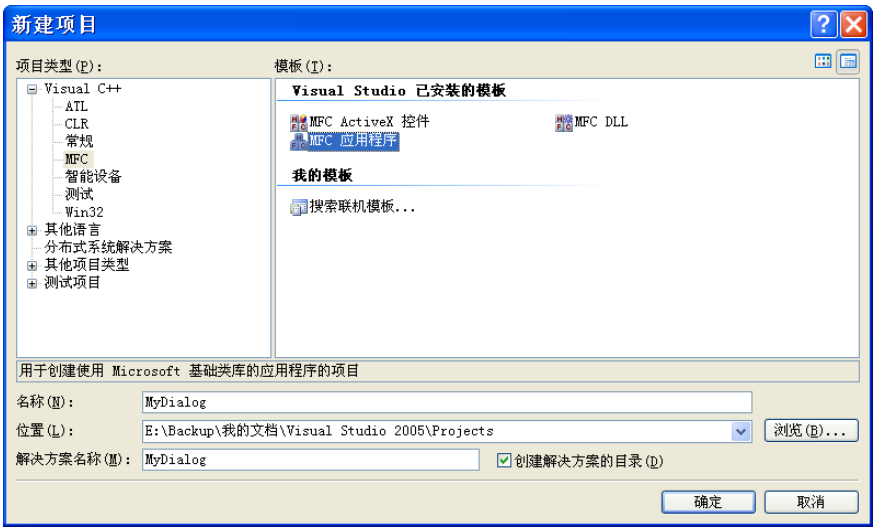


图 11-1　新建项目

选择项目类型为 MFC，选择模板为 MFC 应用程序，在名称里输入“MyDialog”，点击“确定”按钮。然后会出现应用程序向导界面，第一个界面没有需要修改的内容，直接点击“下一步”按钮，弹出界面如图11-2所示。

图 11-2　应用程序向导

VC++ 2005 可以创建多种类型的应用程序，如单文档、多文档和对话框类型的程序，选择“基于对话框”的类型。MFC 的使用是指应用程序引用 API 函数的方式，为了今后可以方便地发布用户自己编写的程序，选择“在静态库中使用 MFC”。继续点击“下一步”按钮进行其他设置，也可以点击“完成”按钮，那么以后的设置均采用默认值。

至此，已经完成了一个 Windows MFC 应用程序的创建及设置了。

11.1.2　应用程序的运行

选择“调试”→“启动调试”菜单项，会弹出启动编译对话框，如图11-3所示。

系统提示是否重新生成项目，为了今后调试程序方便，可以勾选“不再显示此对话框”，由于是新创建的项目，还没有生成可执行文件，点击“是”按钮。

此时可以看到有一个对话框程序运行起来，对话框的标题就是“MyDialog”，上面拥有两个按钮及一段文字，如图11-4所示。

图 11-3　启动编译对话框

图 11-4　Windows 应用程序的运行

此时得到了一个没有任何代码的可运行程序，这些都是 Windows 应用程序向导在发挥作用，以后用户可添加自己的功能代码到其中，可以逐步完善该程序。

11.1.3　应用程序类和源文件

使用 MFC 应用程序向导生成对话框应用程序的基本框架时将派生 3 个类：CmyDialogApp、CMyDialogDlg 和 CaboutDlg，如图11-5所示。

图 11-5　应用程序类视图

主要的程序任务均在 CMyDialogApp 和 CMyDialogDlg 类中实现，此外 MFC 应用程序向导为这两个类生成各自的头文件及实现文件。

MyDialog 程序的应用程序类称为 CMyDialogApp，该类是从 CWinApp 类派生的。CMyDialogApp 的头文件为 MyDialog.h，实现文件为 MyDialog.cpp。应用程序类控制应用程序的所有对象并完成应用程序的初始化工作和最后的清除工作。每个基于 MFC 类库的应用程序都必须有一个而且仅有一个从 CWinApp 类派生的对象。以下是 CMyDialogApp 头文件的代码。

```
// MyDialog.h : PROJECT_NAME 应用程序的主头文件//
#pragma once
#ifndef __AFXWIN_H__
    #error "在包含此文件之前包含“stdafx.h”以生成 PCH 文件"
#endif
#include "resource.h"       // 主符号//
CMyDialogApp:
//有关此类的实现，请参阅 MyDialog.cpp//
class CMyDialogApp : public CWinApp
{
public:
    CMyDialogApp();
//重写
    public:
    virtual BOOL InitInstance();
//实现
    DECLARE_MESSAGE_MAP()
};
extern CMyDialogApp theApp;
```

MyDialog 程序的主程序窗口类称为 CmyDialogDlg。该类是从 CDialog 类派生而来，CMyDialogDlg 的头文件为 MyDialogDlg.h，程序文件为 MyDialogDlg.cpp。对话框经常被使用，因为对话框可以从模板创建，而对话框模板是可以使用资源编辑器方便地进行编辑的。以下是 CMyDialogDlg 的头文件的代码。

```
// MyDialogDlg.h:头文件//
#pragma once
// CMyDialogDlg 对话框
class CMyDialogDlg : public CDialog
{
// 构造
public:
CMyDialogDlg(CWnd* pParent = NULL);                    //标准构造函数
```

```
// 对话框数据
    enum { IDD = IDD_MYDIALOG_DIALOG };
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);
//DDX/DDV 支持
protected:
    HICON m_hIcon;  // 生成的消息映射函数
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    DECLARE_MESSAGE_MAP()};
```

除了生成主要类的源文件外，MFC 应用程序向导还生成为建立 Windows 程序所必须的其他文件。文件视图如图11-6所示。

(1) Resource.h 是标准的头文件，包含所有资源符号的定义。

(2) StdAfx.h 和 StdAfx.cpp 用于生成预编译的头文件。

(3) MyDialog.rc 是包含资源描述信息的资源文件，列有所有的应用程序资源，包括存储在子目录 res 中的图标位图和光标。

(4) MyDialog.rc2 包含不是由 DeveloperStudio 编辑的资源，可以将所有不能由资源编辑器编辑的资源放置到这个文件中。

(5) MyDialog.ico 是包含对话框窗口图标的图标文件。

此外 MFCAppWizard 还生成 ReadMe.txt 文件用于描述为应用程序生成的所有源文件。

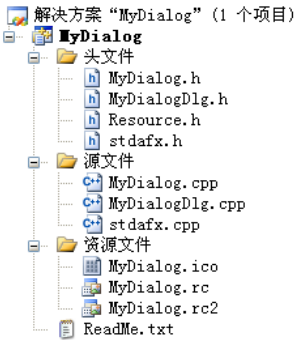


图 11-6 应用程序文件视图

11.1.4 应用程序的控制流程

Windows 应用程序的初始化、运行和结束工作都是由应用程序类完成的。应用程序类构成了应用程序的主执行线程。每个基于 MFC 类库而建立的应用程序都必须有一个且只有一个从 CWinApp 类派生的类对象，该对象在窗口创建之前构造。

与所有的 Windows 应用程序一样，基于 MFC 类库而建立的应用程序也有一个 WinMain 函数。但是，在应用程序中不用编写 WinMain 代码，它是由 MFC 类库提供的，在应用程序启动时调用这个函数。WinMain 函数执行注册窗口类等标准服务，然后再调用对象中的成员函数来初始化和运行应用程序。

MyDialog 程序中关于 CMyDialogApp 类对象的定义是在文件 MyDialog.cpp 中。

```
//唯一的一个 CMyDialogApp 对象
CMyDialogApptheApp;
```

由于 CMyDialogApp 类对象是全局定义的，因此在程序入口函数 WinMain 接收控制之前将调用构造函数。初始时，由 MFC 应用程序向导生成的 CMyDialogApp 构造函数是空的构造函数。因此，编译器将调用基类 CWinApp 的缺省构造函数。CWinApp 构造函数把应用程序对象 CMyDialogApp 的地址保存在一个全局指针中，通过全局指针就可以调用 CMyDialogApp 的成员函数。

在所有全局对象创建之后，WinMain 函数接收控制，在初始化应用程序时，WinMain 函数调用应用程序对象的 InitApplication 和 InitInstance 成员函数。在运行应用程序的消息循环时，WinMain 函数将调用 Run 成员函数。在程序结束时 WinMain 函数将调用应用程序对象的 ExitInstance 成员函数。

成员函数 InitInstance、Run、ExitInstance 和 OnIdle 都是可以覆盖的，其中 InitInstance 是唯一一个必须覆盖的 CWinApp 成员函数，也就是说大多数情况下用户可以不用编写其他函数，直接调用基类的同名函数即可。

由于 Windows 的驱动方式是事件驱动，即程序的流程不是由事件的顺序来控制，而是由事件的发生顺序来控制，所有的事件是无序的，作为一个程序员，在编写程序时，并不知道用户会先按下哪个按纽，也就不知道程序先触发哪个消息。因此程序员的主要任务就是对正在开发的应用程序要发出的或要接收的消息进行排序和管理。事件驱动程序设计是密切围绕消息的产生与处理而展开的，消息是关于发生的事件的消息。

关于 Windows 消息循环的更多知识，在此不再介绍，请读者参考相关书籍。

11.2　调用Windows公共对话框的实例

在 Windows 应用程序设计中，经常会接触到几种常见的操作，例如，打开一个文件对话框或是打印设置对话框等，MFC 类库中提供了对应的常用对话框来完成该操作，以减少程序员开发的工作量，详见表 11-1。

表 11-1　Windows 公共对话框

对话框类名	用途
CcolorDialog	选择一种颜色
CfileDialog	选择打开或保存的文件名
CFindReplaceDialog	在文本文件中打开查找或替换对话框
CFontDialog	打开字体对话框
CPrintDialog	打开打印对话框

在下面的例子中调用了上面的几个公共对话框中的颜色、文件，以及字体对话框。

11.2.1　使用对话框编辑器

按照上面创建的工程，选择“视图”→“资源视图”菜单项，打开资源对话框，如图11-7所示。

图 11-7　“资源视图”对话框

在资源窗口双击对话框文件夹下面的 IDD_MYDIALOG_DIALOG。打开对话框编辑器后，显示程序的主窗体界面，如图11-8所示。

选择窗体上面的标签控件，并删除，选择“视图”→“工具箱”菜单项，打开工具箱，用鼠标选择“Button”控件拖入对话框上面，并对按钮进行属性设置。

修改属性 caption 为“文件对话框”，修改属性 ID 为“IDC_BUTTON_FILE”，如图11-9所示。

图 11-8 对话框编辑器界面

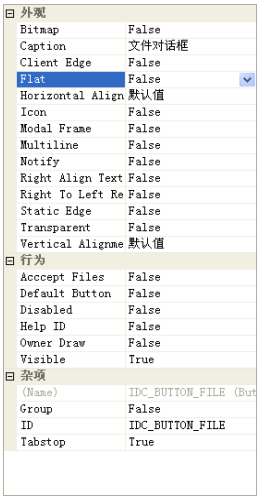


图 11-9 控件属性设置

再拖入两个按钮，修改属性 caption 为“颜色对话框”和“字体对话框”，修改属性 ID 为“IDC_BUTTON_COLOR”和“IDC_BUTTON_FONT”。

此时主对话框界面如图11-10所示。

图 11-10 对话框编辑结果

11.2.2 编写代码

在对话框编辑器中双击文件对话框按钮，系统会自动添加消息函数的声明和实现到相应的头文件和实现文件中去，并添加消息映射。以下是空的消息函数代码。

```
void CMyDialogDlg::OnBnClickedButtonFile()
{
// TODO: 在此添加控件通知处理程序代码
}
```

用户在指定位置添加代码就可以了。下面简单介绍 CFileDialog 的构造函数用法。

```
CFileDialog(BOOL bOpenFileDialog,LPCTSTR lpszDefExt = NULL,
LPCTSTR lpszFileName = NULL,
DWORD dwFlags = OFN_HIDEREADONLY | FN_OVERWRITEPROMPT,
LPCTSTR lpszFilter = NULL, CWnd* pParentWnd = NULL,
DWORD dwSize = 0,BOOL bVistaStyle = TRUE);
```

bOpenFileDialog = true 时，对话框是“打开文件对话框”；

bOpenFileDialog = false 时，对话框是“另存为对话框”；

以后的函数参数均采用默认值即可。

在该消息函数中添加如下代码：

```
void CMyDialogDlg::OnBnClickedButtonFile()
{   // TODO: 在此添加控件通知处理程序代码
CString strtmp;          //声明一个字符串对象
CFileDialog cf(true);    //声明一个对话框对象
if(cf.DoModal()==IDOK)  //打开对话框，判断返回值
{   strtmp=cf.GetPathName();    //点击打开文件，给字符串赋值为文件名字
}
else
{   strtmp="没有选择文件!";        //单击取消或是直接关闭文件对话框
}
AfxMessageBox(strtmp);           //利用消息框弹出信息字符串
}
```

运行程序，点击“文件对话框”按钮，选择磁盘上任意一个文件，单击“打开”按钮，如图11-11所示。

点击“文件对话框”按钮后，直接关闭或取消，将出现如图11-12所示画面。

图 11-11　选择一个文件后显示的消息框

图 11-12　没有选择文件显示的消息框

再利用相同的方法，在颜色对话框及字体对话框按钮事件中输入如下代码。运行程序可以显示颜色及字体对话框。

```
void CMyDialogDlg::OnBnClickedButtonColor()
{   // TODO: 在此添加控件通知处理程序代码
    CString strtmp;                  //声明一个字符串对象
    CColorDialog cc;                 //声明颜色对话框
    if(cc.DoModal() == IDOK )        //打开对话框，判断返回值
    {       strtmp="选择了特定颜色";  //单击确定按钮，返回指定颜色
    }
    else
    {       strtmp="没有选择颜色";    //单击取消或是直接关闭颜色对话框
    }
    AfxMessageBox(strtmp);           //利用消息框弹出信息字符串
}

void CMyDialogDlg::OnBnClickedButtonFont()
{   // TODO: 在此添加控件通知处理程序代码
    CString strtmp;                      //声明一个字符串对象
    CFontDialog cf;                      //声明字体对话框
    if(cf.DoModal() == IDOK )            //打开对话框，判断返回值
```

```
        {       strtmp=cf.GetFaceName();    //单击确定按钮，返回指定字体名称
        }
        else
        {       strtmp="没有选择字体";        //单击取消或是直接关闭颜色对话框
        }
        AfxMessageBox(strtmp);              //利用消息框弹出信息字符串
    }
```

程序代码中注释内容比较多，就不再一一讲解了，用户可以快速利用 MFC 里面提供的一些标准对话框，对自己的程序进行更多的设置。

11.3 利用VC++ 2005 连接数据库实例

随着 VC++ 2005 软件开发工具的广泛推广，对数据库方面的应用日趋广泛和深入，越来越多的软件开发人员和爱好者希望理解并掌握应用 VC++ 2005 管理开发数据库的技术和方法。VC++ 2005 提供了几种接口（ODBC、DAO、OLE/DB、ADO）来支持数据库编程，利用这些接口可以在程序中直接操作各种各样的数据库，如（SQL Server、Microsoft Access、Microsoft FoxPro）等。

下面介绍利用 DAO 访问 Access 数据库的例子。

11.3.1 建立工程DAOAccess

继续按照前面的例子，再次建立名为“DAOAccess”的新工程，注意还是选择“基于对话框的程序”。为了方便学习使用，暂时先不考虑字符集的问题，把“使用 Unicode”勾选取消掉。

11.3.2 建立Access文件

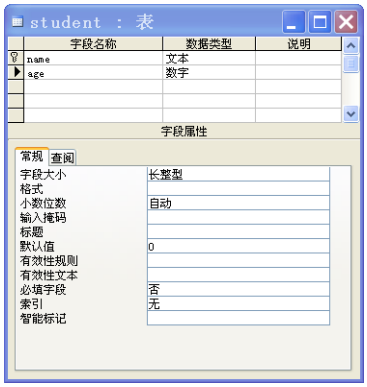


图 11-13 Access 数据库中表字段设置

打开 Office 里面的 Access 软件建立一个数据库文件 DAOAccess.mdb，保存在上述工程文件夹中，内有数据表 student，如图11-13所示，有两个字段（name, age），其中 name 为主键字段，可以提前输入几条实验记录数据。

11.3.3 修改主窗体界面

打开对话框编辑器，删除主窗体上的所有控件，并按表 11-2 添加控件。

表 11-2 用于连接数据库的控件

控件类型	控件 ID	备注
List Ctrl	IDC_LIST_STUDENT	属性 view 设置为 Report
Static Text	IDC_STATIC	属性 caption 设置为“姓名”
Static Text	IDC_STATIC	属性 caption 设置为“年龄”
Edit Ctrl	IDC_EDIT_NAME	
Edit Ctrl	IDC_EDIT_AGE	
Button	IDC_BUTTON_ADD	属性 caption 设置为“增加”
Button	IDC_BUTTON_DELETE	属性 caption 设置为“删除”
Button	IDC_BUTTON_CLOSE	属性 caption 设置为“关闭”

利用鼠标调整控件大小及位置，如图11-14所示。

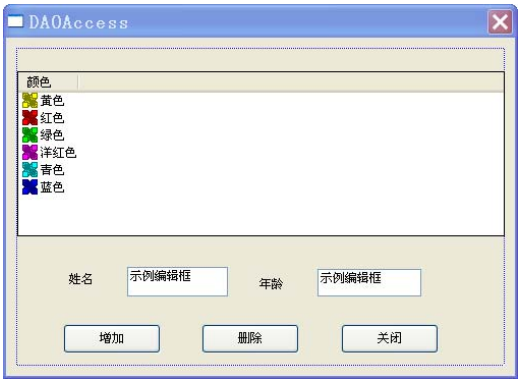


图 11-14　DAOAccess 对话框

11.3.4　添加代码

（1）首先应确保包含了 afxdao.h 头文件，可以在 StdAfx.h 文件中包含它：

```
#include afxdao.h      //加入 DAO 数据库支持
```

（2）声明 DAO 库及其记录集变量，可在实现文件 DAOAccessDlg.cpp 的开始位置加入下面代码：

```
CDaoDatabase db;                    //数据库
CDaoRecordset RecSet(db);           //记录集
```

（3）添加 InsertData 函数

在 DAOAccessDlg.h 中添加如下代码，声明函数 InsertData：

```
void InsertData(CString strName, CString strAge);
```

在 DAOAccessDlg.cpp 中添加如下代码，实现函数 InsertData：

```
void CDAOAccessDlg::InsertData(CString strName,CString strAge)
{
    CListCtrl * pl;//列表控件指针变量
    pl=((CListCtrl *)(GetDlgItem(IDC_LIST_STUDENT)));  //获取列表控件指针

    int i=pl->GetItemCount();              //获得列表当前保存的条目数量
    pl->InsertItem(i,strName);          //在最后插入一个新条目
    pl->SetItemText(i,1,strAge);        //设置该新条目的下标为列内容，即年龄
}
```

GetDlgItem()函数可以由控件 ID 获得控件指针，这是 Windows 可视化编程方式下获取控件指针接口最常见的方式，上面例子中将这个指针强制转换成一个列表控件指针，并保存下来以便以后使用。

实现完成 InsertData 函数后，可以逐步添加代码，每一步读者都可以自行检验效果。

（4）修改 OnInitDialog()函数

在 DAOAccessDlg.cpp 中的 OnInitDialog()中添加如下代码，以完成列表控件初始化设置，打开数据库和记录集，自动读取所有记录，插入记录到列表中。

```
BOOL CDAOAccessDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ……//系统生成代码
    //TODO: 在此添加额外的初始化代码
    CListCtrl * pl;//列表控件指针变量
    pl=((CListCtrl *)(GetDlgItem(IDC_LIST_STUDENT)));//获取列表控件指针
    pl->InsertColumn(0,"学生姓名",0,100);//添加列表控件的列头，位置下标为0，
    //内容是"学生姓名"，对齐方式是左对齐，宽度为100个像素
    pl->InsertColumn(1,"学生年龄",0,100);//添加列表控件的列头，位置下标为1，
    //内容是"学生年龄"，对齐方式是左对齐，宽度为100个像素
    COleVariant var;    //字段类型
    CString strName,strAge,strFile;
    CString strSql;
    strSql="SELECT * FROM student"; //读取数据库表中所有记录的sql语句
    strFile = "DAOAccess.mdb";      //数据库文件的名字
    db.Open(strFile);   //打开已创建的DAOAccess数据库
    RecSet.Open(AFX_DAO_USE_DEFAULT_TYPE,strSql,NULL);//打开student表
    while(!RecSet.IsEOF())//当记录集指针到达结尾的时候，该函数返回真，否则为假
    {
        RecSet.GetFieldValue("name",var);   //读取当前记录的name字段
        strName =(LPCTSTR)var.bstrVal;
        RecSet.GetFieldValue("age",var);    //读取当前记录的age字段
        strAge.Format("%d",var.intVal);
        InsertData(strName,strAge); //在列表控件上插入刚刚读取的记录
        RecSet.MoveNext();          //记录指针向后移动一个
    }
    return TRUE; //除非将焦点设置到控件，否则返回TRUE
}
```

(5) 添加“增加”、“删除”、“关闭”按钮事件函数代码

在对话框编辑器中，双击指定的按钮，添加如下代码：

```
void CDAOAccessDlg::OnBnClickedButtonAdd()
{
    // TODO: 在此添加控件通知处理程序代码
    CString strName,strAge;
    GetDlgItem(IDC_EDIT_NAME)->GetWindowText(strName);
    //获取Name编辑框控件上的内容，前面是获得控件指针，GetWindowText函数可以获
    //得指定控件上面的文本内容，所有控件均可以调用这个函数
    GetDlgItem(IDC_EDIT_AGE)->GetWindowText(strAge);
    //获取Age编辑框控件上的内容
    RecSet.AddNew();//记录集开始添加新记录
    RecSet.SetFieldValue("Name",(LPCTSTR)strName);//设置新记录的Name字段
    RecSet.SetFieldValue("Age",(LPCTSTR)strAge);  //设置新记录的Age字段
    RecSet.Update();//保存记录集
    InsertData(strName,strAge);//在控件上插入新输入的记录
}
```

```
void CDAOAccessDlg::OnBnClickedButtonDelete()
{
    // TODO: 在此添加控件通知处理程序代码
    CListCtrl * pl;          //列表控件指针变量
    pl=((CListCtrl *)(GetDlgItem(IDC_LIST_STUDENT)));    //获取列表控件指针
    int nIndex = pl->GetNextItem( -1, LVNI_SELECTED );
    //利用列表控件成员函数获得选中的条目索引号

    RecSet.MoveFirst();     //移动记录集指针到第一个位置
    RecSet.Move(nIndex);    //向后移动 nIndex
    RecSet.Delete();        //删除当前记录集
    pl->DeleteItem(nIndex); //删除当前选中条目
}

void CDAOAccessDlg::OnBnClickedButtonClose()
{
    // TODO: 在此添加控件通知处理程序代码
    RecSet.Close();         //关闭记录集
    db.Close();             //关闭数据库
    OnOK();                 //关闭对话框
}
```

程序最终运行效果如图11-15所示。

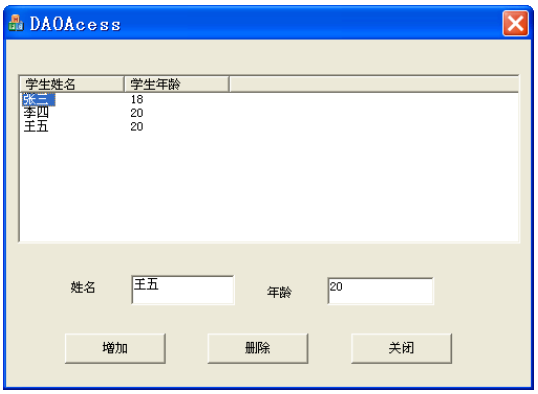


图 11-15　DAOAccess 运行效果

程序代码中注释部分较多，就不再一一讲解。这个读写数据库的程序，距离完美操作数据库还有较大的距离，只是给读者展示一下如何利用 Visual C++ 2005 来操作数据库。

附录A　ASCII码表

八进制	十六进制	十进制	字符	八进制	十六进制	十进制	字符
00	00	0	nul	100	40	64	@
01	01	1	soh	101	41	65	A
02	02	2	stx	102	42	66	B
03	03	3	etx	103	43	67	C
04	04	4	eot	104	44	68	D
05	05	5	enq	105	45	69	E
06	06	6	ack	106	46	70	F
07	07	7	bel	107	47	71	G
10	08	8	bs	110	48	72	H
11	09	9	ht	111	49	73	I
12	0a	10	lf	112	4a	74	J
13	0b	11	vt	113	4b	75	K
14	0c	12	ff	114	4c	76	L
15	0d	13	cr	115	4d	77	M
16	0e	14	so	116	4e	78	N
17	0f	15	si	117	4f	79	O
20	10	16	dle	120	50	80	P
21	11	17	dc1	121	51	81	Q
22	12	18	dc2	122	52	82	R
23	13	19	dc3	123	53	83	S
24	14	20	dc4	124	54	84	T
25	15	21	nak	125	55	85	U
26	16	22	syn	126	56	86	V
27	17	23	etb	127	57	87	W
30	18	24	can	130	58	88	X
31	19	25	em	131	59	89	Y
32	1a	26	sub	132	5a	90	Z
33	1b	27	esc	133	5b	91	[
34	1c	28	fs	134	5c	92	\
35	1d	29	gs	135	5d	93	]
36	1e	30	rs	136	5e	94	^
37	1f	31	us	137	5f	95	_
40	20	32	space	140	60	96	'
41	21	33	!	141	61	97	a
42	22	34	"	142	62	98	b
43	23	35	#	143	63	99	c

（续表）

八进制	十六进制	十进制	字符	八进制	十六进制	十进制	字符
44	24	36	$	144	64	100	d
45	25	37	%	145	65	101	e
46	26	38	&	146	66	102	f
47	27	39	`	147	67	103	g
50	28	40	(	150	68	104	h
51	29	41	)	151	69	105	i
52	2a	42	*	152	6a	106	j
53	2b	43	+	153	6b	107	k
54	2c	44	,	154	6c	108	l
55	2d	45	-	155	6d	109	m
56	2e	46	.	156	6e	110	n
57	2f	47	/	157	6f	111	o
60	30	48	0	160	70	112	p
61	31	49	1	161	71	113	q
62	32	50	2	162	72	114	r
63	33	51	3	163	73	115	s
64	34	52	4	164	74	116	t
65	35	53	5	165	75	117	u
66	36	54	6	166	76	118	v
67	37	55	7	167	77	119	w
70	38	56	8	170	78	120	x
71	39	57	9	171	79	121	y
72	3a	58	:	172	7a	122	z
73	3b	59	;	173	7b	123	{
74	3c	60	<	174	7c	124	|
75	3d	61	=	175	7d	125	}
76	3e	62	>	176	7e	126	~
77	3f	63	?	177	7f	127	del

附录B　习 题 答 案

第 1 章　习题答案

一、选择题 ACC

二、填空题

(1) 机器语言指令

(2) 结构化的程序设计，面向对象的程序设计

(3) 类

第 2 章　习题答案

一、选择题 ADDBB　BABBC　BADCA

二、填空题

(1)10　(2)12.5　(3)x<2　(4)表达式具有值，而语句是没有值的并且语句末尾要加分号 (5)右 左　(6)x*(y+8)　(7)27　(8)14　(9)35　(10)103

三、问答题

(1) A1 = 2+3 + 1 + 4 = 5+1+4=6+4=10；A2 = 2 + (3+2)/(2+4) = 2 + 5/6 = 2 + 0 = 2

(2) 算术运算符由加'+'减'-'乘'*'除'/'和取余'%'组成

(3) 依次为'*'、'+'、'>'、 '>='、 '&'、'&&'、 '*='

(4) '-'可以用于代表双目运算符减运算，同时可以代表单目运算符使后面的操作数变换符号。'&'可以用于代表双目运算符按位与运算，同时可以是地址运算符，表示后面操作数的地址。'*'可以用于代表双目运算符乘运算，同时可以与'&'相对应的单目运算符，表示后面的操作数地址中的值

(5) '+'、'&'作为单目运算符时结合性和从右到左，而作为双目运算符时，则为从左到右。'='运算符的结合性为从右到左，'||'运算符的结合性为从左到右

(6) 3 ；6.0；4.5；644；188；238

第 3 章　习题答案

一、选择题　DDADDB

二、填空题

(1) do-while 语句是先执行循环体，然后检查循环条件；while 语句是先检查循环条件，再执行循环体

(2) 10，20，10

(3) 语句，执行程序语句

(4) break 语句用在循环语句的循环体内的作用是终止当前的循环语句

(5) 根据程序的目的，有时需要程序在满足另一个特定条件时跳出本次循环

第 4 章　习题答案

一、选择题 BCAAD

二、判断题 √×√√×

三、阅读程序，写出运行结果

(1) dlrow, olleh (2) 9 (3) ABCDEFG I (4) 24

第 5 章 习题答案

一、选择题 ADABD

二、判断题 √××√√√×

三、填空题

(1) *p &p (2) 变量的值和指针都未发生变化 (3) 25 (4) *(a+i) (5) *p (6) 5

四、(1) 0012FF3C 0012FF3C

0 0

4 4 40

(2) 14 16 10 12

11 39 13 25

(3) 7 8 6 10 3

(4) 0 1 1 2 3

5 8 13 21 34

55 89

(5) 267 53.4

第 6 章 习题答案

一、选择题 DCBBDA

二、判断题 √××√√√×

三、写出程序的运行结果或功能

(1) 10，20

10，20 ，5

(2) 用递归方法求两个整数的最大公约数

(3) 用非递归方法求两个整数的最大公约数

(4) 5 4 3 2 1 0

0

(5) 将一个整数从个位数开始分解并输出

第 7 章 习题答案

一、填空题

(1) 结构体变量名.成员名 (2) (*p).y (3) 10 (4) 3

二、分析题

(1) 40 8

(2) 第七行没有分号。初始化应该写成 stu1={"20100001","胡明", {10,15,1988}};

第 8 章 习题答案

一、选择题 CBABC

二、分析程序的运行结果，并加以解释

(1)

x=0, y=0

x=10, y=15

node(10, 15)被撤销

node(0, 0)被撤销

(2)

node(3, 10)

node(5, 10)

第 9 章　习题答案

一、选择题 ACBDC

三、分析以下程序的执行结果

构造基类

n=1

构造派生类

m=2

析构派生类

析构基类

第 10 章　习题答案

一、选择题 CDBD

附录C　常用库函数

1．数学函数

函数名称	函数原型	数学表示	功能说明
整数绝对值函数	int abs(int i)	$\|i\|$	返回参数 *i* 的绝对值
实数绝对值函数	double fabs(double x)	$\|x\|$	返回实数 *x* 的绝对值
正弦函数	double sin(double x)	sin*x*（*x* 为弧度）	返回弧度为 *x* 的正弦值
余弦函数	double cos(double x)	cos*x*（*x* 为弧度）	返回弧度为 *x* 的余弦值
正切函数	double tan(double x)	tan*x*（*x* 为弧度）	返回弧度为 *x* 的正切值
平方根函数	double sqrt(double x)	$\sqrt{x}$（$x\geqslant 0$）	返回 *x* 的算数平方根
指数函数	double exp(double x)	e^x（e=2.71828）	返回 e^x 的值
幂函数	double pow(double x, double y)	*xy*	返回 *xy* 的值
自然对数函数	double log(double x)	ln*x*　（*x*>0）	返回以 e 为底 *x* 的对数
对数函数	double log10(double x)	$\log_{10}x$　（*x*>0）	返回以 10 为底 *x* 的对数
符号函数	int sgn(double x)	sgn(*x*)	*x*>0 返回 1 *x*=0 返回 0 *x*<0 返回−1
向上取整函数	double ceil(double x)	$\lceil x \rceil$	返回大于等于 *x* 的最小整数
向下取整函数	double floor(double x)	$\lfloor x \rfloor$	返回小于等于 *x* 的最大整数
随机函数	int rand(x)		返回 0～32767 之间的整数
改变随机数序列	Void srand(int x)		生成与 *x* 对应的随机序列
终止程序运行	void exit(int status)		通常参数为 0 表示正常结束，非 0 表示不正常结束

上述函数表中，除后 3 个函数之外，均为数学函数，它们的函数原型包含在系统建立的 math 或 cmath 头文件中，后 3 个为常用的一般函数，它们的函数原型包含在系统建立的 stdlib 或 cstdlib 头文件中。

2．常用反函数公式

函数	导出式
Secant	sec(*x*) = 1 / cos(*x*)
Cosecant	cosec(*x*) = 1 / sin(*x*)
Cotangent	cotan(*x*) = 1 / tan(*x*)
Inverse Sine	arcsin(*x*) = atn(*x* / sqr(-*x* * *x* + 1))
Inverse Cosine	arccos(*x*) = atn(-*x* / sqr(-*x* * *x* + 1)) + 2 * atn(1)
Inverse Secant	arcsec(*x*) = atn(*x* / sqr(*x* * *x* - 1)) + sgn((*x*) -1) * (2 * atn(1))
Inverse Cosecant	arccosec(*x*) = atn(*x* / sqr(*x* * *x* - 1)) + (sgn(*x*) - 1) * (2 * atn(1))
Inverse Cotangent	arccotan(*x*) = atn(*x*) + 2 * atn(1)
Hyperbolic Sine	hsin(*x*) = (Exp(*x*) - Exp(-*x*)) / 2
Hyperbolic Cosine	hcos(*x*) = (Exp(*x*) + Exp(-*x*)) / 2

（续表）

函数	导出式
Hyperbolic Tangent	htan(*x*) = (Exp(*x*) - Exp(-*x*)) / (Exp(*x*) + Exp(-*x*))
Hyperbolic Secant	hsec(*x*) = 2 / (Exp(*x*) + Exp(-*x*))
Hyperbolic Cosecant	hcosec(*x*) = 2 / (Exp(*x*) - Exp(-*x*))
Hyperbolic Cotangent	hcotan(*x*) = (Exp(*x*) + Exp(-*x*)) / (Exp(*x*) - Exp(-*x*))
Inverse Hyperbolic Sine	harcsin(*x*) = Log(*x* + Sqr(*x* * *x* + 1))
Inverse Hyperbolic Cosine	harccos(*x*) = Log(*x* + Sqr(*x* * *x* - 1))
Inverse Hyperbolic Tangent	harctan(*x*) = Log((1 + *x*) / (1 - *x*)) / 2
Inverse Hyperbolic Secant	harcsec(*x*) = Log((Sqr(-*x* * *x* + 1) + 1) / *x*)
Inverse Hyperbolic Cosecant	harccosec(*x*) = Log((Sgn(*x*) * Sqr(*x* * *x* + 1) +1) / *x*)
Inverse Hyperbolic Cotangent	harccotan(*x*) = Log((*x* + 1) / (*x* - 1)) / 2
Logarithm to base N	logn(*x*) = Log(*x*) / Log(*n*)

3．与String有关的函数表

名称	函数格式	功能

string()	string();
	string(size_type length, char ch);
	string(const char *str);
	string(const char *str, size_type length);
	string(string &str, size_type index, size_type length);
	string(input_iterator start, input_iterator end);
		创建一个新字符串

append()	basic_string &append(const basic_string &str);
	basic_string &append(const char *str);
	basic_string &append(const basic_string &str, size_type index, size_type len);
	basic_string &append(const char *str, size_type num);
	basic_string &append(size_type num, char ch);
	basic_string &append(input_iterator start, input_iterator end);
		在字符串的末尾添加字符

assign()	basic_string &assign(const basic_string &str);
	basic_string &assign(const char *str);
	basic_string &assign(const char *str, size_type num);
	basic_string &assign(const basic_string &str, size_type index, size_type len);
	basic_string &assign(size_type num, char ch);
		为字符串赋值

at()	reference at(size_type index);	返回一个引用
begin()	iterator begin();	返回指向字符串的第一个元素
c_str()	const char *c_str();	返回一个指向字符串的指针
capacity()	size_type capacity();	返回字符串中字符的个数

compare()	int compare(const basic_string &str);
	int compare(const char *str);
	int compare(size_type index, size_type length, const basic_string &str);
	int compare(size_type index, size_type length, const basic_string &str, size_type index2,size_type length2);
	int compare(size_type index, size_type length, const char *str, size_type length2);
		返回值情况：
		小于零 this<str
		零 this==str
		大于零 this> str

copy()	size_type copy(char *str, size_type num, size_type index);	复制 num 个字符到 str 中

（续表）

名称	函数格式	功能
data()	const char *data();	返回指向自己的首字符的指针
empty()	bool empty();	判断字符串是否为空
end()	iterator end();	判断是否指向字符串末尾

erase()	iterator erase(iterator pos);
	iterator erase(iterator start, iterator end);
	basic_string &erase(size_type index = 0, size_type num = npos);
		删除字符

find()	size_type find(const basic_string &str, size_type index);
	size_type find(const char *str, size_type index);
	size_type find(const char *str, size_type index, size_type length);
	size_type find(char ch, size_type index);
		返回 str 在字符串中第一次出现的位置

insert()	iterator insert(iterator i, const char &ch);
	basic_string &insert(size_type index, const basic_string &str);
	basic_string &insert(size_type index, const char *str);
	basic_string &insert(size_type index1, const basic_string &str, size_type index2, size_type num);
	basic_string &insert(size_type index, const char *str, size_type num);
	basic_string &insert(size_type index, size_type num, char ch);
	void insert(iterator i, size_type num, const char &ch);
	void insert(iterator i, iterator start, iterator end);
		在指定位置前插入一个字符 ch 或字符串 str

length()	size_type length();	返回字符串的长度
max_size()	size_type max_size();	返回字符串能保存的最大字符数

replace()	basic_string &replace(size_type index, size_type num, const basic_string &str);
	basic_string &replace(size_type index1, size_type num1, const basic_string &str, size_type index2,size_type num2);
	basic_string &replace(size_type index, size_type num, const char *str);
	basic_string &replace(size_type index, size_type num1, const char *str, size_type num2);
	basic_string &replace(size_type index, size_type num1, size_type num2, char ch);
	basic_string &replace(iterator start, iterator end, const basic_string &str);
	basic_string &replace(iterator start, iterator end, const char *str);
	basic_string &replace(iterator start, iterator end, const char *str, size_type num);
	basic_string &replace(iterator start, iterator end, size_type num, char ch);
		用指定个数的字符替换原来的字符

reserve()	void reserve(size_type num);	设置本字符串的容量以保留 num 个字符空间
resize()	void resize(size_type num); void resize(size_type num, char ch);	重新设置字符空间的大小
size()	size_type size();	返回字符串中的字符数
substr()	basic_string substr(size_type index, size_type num = npos);	返回从 index 开始，长 num 个字符的字串
swap()	void swap(basic_string &str);	交换字符串

注意：at()函数返回指向 index 位置的字符的一个引用，如果 index 不在字符串范围内，at()将报告"out of range"错误，并抛出 out_of_range 异常。

由于本表中出现的参数太多，请读者根据参数字面含义自行上机试运行，或参考其他相关文献的说明。

附录D　程序调试与异常处理

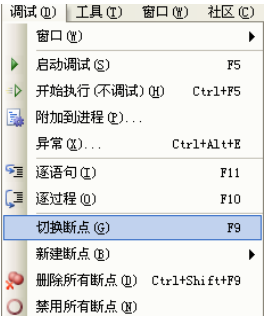
1．程序调试

在程序编写过程中，难免会出现一些错误。为了解决这些错误，程序员需要对应用程序进行调试，查出错误产生的原因。Visual Studio C++ 2005 中提供了调试器。主要步骤如下。

设置断点。所谓断点就是一个信号，它通知调试器在断点处停止运行。发生中断时，程序和调试器处于中断状态。进入中断状态时并不会终止或结束程序的执行，所有程序都留在内存中，并可在任何时候继续。

设置断点有三种方法：在需要设置断点的行旁边单击；右键单击设置断点的代码行，选择“断点”→“插入断点”，如图(a)所示；单击要设置断点的行代码，选择菜单中的“调试”→“切换断点(G)”命令，如图(b)所示。

图(a)　右键插入断点



图(b)　菜单插入断点

设置断点后，就会在设置断点的行旁边的灰色空白处出现一个红色圆点，并且该行代码也高亮显示。

删除断点有四种方式：单击断点的行旁边的灰色空白处的红色圆点；右键单击断点行旁边的灰色空白处的红色圆点，选择“删除断点”；右键单击设置断点的行代码，选择“断点”→“删除断点”；单击要设置断点的行代码，选择菜单中的“调试”→“切换断点(G)”命令。

(1) 如何使用开始、中断和停止执行

调试程序时，可以通过使用开始、中断和停止执行功能随时控制代码段的执行状态，操作步骤如下。

① 开始执行。

可以通过在菜单“调试”→“启动调试”→“逐语句”或“逐过程”来执行程序并调试，也可以通过右键单击可执行代码中的某行，然后从快捷菜单中选择“运行到光标处”。

如果选择“启动调试”，则应用程序启动并一直运行到断点。也可以在任何时刻中断执行，以检查值，修改变量，或检查程序状态。如果选择了“逐语句”或“逐过程”，则应用程序启动并执行，然后在第一行中断。

如果选择“运行到光标处”，则应用程序启动并一直运行到断点或光标位置。如果光标在

断点前，则程序运行到光标处；如果光标在断点后，则程序运行到断点处。某些情况下，不出现中断，这意味着执行始终未到达设置光标处的代码。

程序调试过程中，当光标指向断点之前的变量或属性时，将会自动在光标的下方弹出一个提示框，并在提示框中显示当前所指变量或属性的值。如果用光标指向断点以后的变量，提示框中只显示变量的初始值或上一次运算的结果。

② 中断执行。

当程序执行到达一个断点或发生异常，调试器就会中断程序的执行。也可以通过“调试”→“全部终止”命令手动中断执行。这时调试器将停止所有在调试器下运行的程序的执行。但程序并不退出，而且可以随时恢复执行。调试器和应用程序现在处于终端模式。

③ 停止执行。

停止执行意味着终止当前正在调试的程序并结束调试会话。与中断执行不同，中断执行意味着暂停正在调试的进程执行，但调试会话仍处于活动状态。

可以通过选择菜单中的“调试”→“停止调试”命令，或单击“调试”工具栏中的按钮来结束运行和调试，也可以退出正在调试的应用程序，调试将自动停止。

(2) 单步执行

单步执行是最常见的调试过程之一，即每次执行一行代码。调试菜单中提供了 2 个单步执行命令，即逐语句、逐过程。

“逐语句”和“逐过程”的差异仅在于它们处理函数调用的方式不同。这两个命令都指示调试器执行下一行的代码。如果某一行包含函数调用，“逐语句”仅执行调用本身，然后在函数内的第一个代码行处停止。而“逐过程”执行整个函数，然后在函数外的第一行处停止。如果要查看函数调用的内容，则使用“逐语句”。若要避免单步执行函数，那么最好使用“逐过程”。

(3) 运行到指定位置

如果在调试过程中，想执行到代码中的某一点，然后中断。可以在要中断的位置设置断点，也可以在“调试”菜单中选择“启动”或“继续”。该位置可以在源窗口或反汇编窗口中设置。

在代码窗口中运行到光标处，可以在代码窗口中右键单击某行，并从快捷菜单中选择“运行到光标处”，如图(c)所示。

图(c)　运行到光标处

在反汇编窗口中运行到光标处，可以在反汇编窗口中右键单击某行，并从快捷菜单中选择“运行到光标处”。如果反汇编窗口没有显示，那么从调试菜单中选择“窗口”→“反汇编”。反汇编窗口只能在中断模式下才能进行查看。

调试程序是编程人员的基本功，学好和用好程序调试可以加快编程的速度及准确性。

2. 异常处理

在程序调试时，一些程序虽然可以通过编译也能运行，但在程序运行中会出现异常，从而得不到正确结果，导致程序非正常终止，这类错误不宜被发现，是程序调试中的难点。C++系统采用了异常处理机制来解决这类问题。

异常处理机制主要由三部分组成：检查(try)、抛出(throw)和捕获(catch)，将需要检查的

语句放在 try 块中，当出现异常时，throw 发出一个异常信息，而用 catch 来捕，如果捕获到异常信息后，就在 catch 块中进行处理。异常处理的基本语句结构的一般使用形式如下：

```
throw 表达式;                    //抛出异常
try                              //检查异常
{
    ...                          //检查异常块
}
 catch(类型 n,参数 n)            //捕获异常,可以有多个 catch 块
    {
    ...
    }
```

异常处理的一般执行过程如下：

① 程序执行到 try 块后，接着执行块内的代码。

② 如果在执行 try 块内代码或在 try 块内的代码中调用任何函数期间没有发生异常，则 try 后的 catch 块将不被执行，程序将从 catch 块后面的语句继续执行。

③ 如果在执行 try 块内代码或在 try 块内的代码中调用任何函数期间有异常抛出，编译器就从抛出异常的本层或上层调用函数中查找一个 catch 块用于捕获该异常。

④ 如果找到一个匹配的 catch 块，说明捕获到此异常，则形参通过 catch 语句中抛出的表达式进行初始化。之后，就销毁 catch 块对应的 try 块开始和抛出异常语句 throw 之间的所有自动变量，然后执行 catch 块中的语句处理异常，最后程序跳到最后一个 catch 块后面的语句继续执行。

⑤ 如果没有找到匹配的 catch 块，则自然终止程序。